Getting Started with Deep Learning

Jacob Johnson
jmjohnson33@wisc.edu

Introduction

This document is used as a helpful guide for getting started with deep learning programming in Keras. Questions or comments should be directed to Jacob Johnson.

Contents

1 Introduction

This guide serves as a practical starting point for writing code in deep learning, specifically using the deep learning Python library Keras. First, some software needs to be installed. Many options are possible. This guide covers a solution that is hopefully simple and accessible without a strong background in programming.

For setting up a Windows workstation, go to section 2.

For setting up a Linux workstation, go to section 3

2 Windows Installation

2.1 Hardware

The use of a dedicated GPU is strongly advised for use in any deep learning work. A CPU is adequate in most cases for making predictions with a trained model (inferencing) but the training process itself can take hours on a good GPU or many days on a plain CPU.

This guide will provide all necessary instructions for installing the CPU-based software, and provide coding examples that are sufficiently simple to be able to run on a modern CPU in a few minutes. To utilize an Nvidia GPU, CUDA and cuDNN must also be installed. The installation varies greatly between Windows and Linux and is documented online so that part is left to you (or your IT administrator).

2.2 Anaconda

This guide makes use of software called Anaconda that simplifies installation and management of Python packages.

Two options are available: the full Anaconda installation, or a lighter version called Miniconda. If hard disk space is a potential issue, then installed Miniconda. Otherwise, installing full Anaconda will combine a lot of these steps into one and save you future installations.

2.2.1 Full Anaconda install

To install Anaconda on Windows, go to anaconda.com/download and click ‘download’ underneath Python 3.6. Save the file to somewhere you can find it, then run the .exe and follow the prompts to install

2.2.2 Miniconda install

To install Miniconda, go to conda.io/miniconda and select the 64-bit, Python 3.6 download for Windows. Save the file to somehwere you can find it, then run the .exe and follow the prompts to install.

2.2.3 Setting up Conda

Once your selected version of Anaconda is installed, go to the start menu and search for “Anaconda Prompt”. Selecting it will open a terminal window. First, setup a conda environment. Type

```
conda create -n (environment-name) python=3.5
```

Replace “(environment-name)” with a memorable name for your conda environment. You will use the name every time you start using Conda. I recommend “python35”, which indicates the version of python that will be installed in that environment.

Once the environment is created, you still need to activate it. Type

```
        activate (environment-name)
```

You will know you are now in your Conda environment because the prompt in the terminal will change to beginning with your envionment name in ( ). Any installations you make now will not affect any other environment that you create. This is also the simplest way of installing Python 3.5, which maximizes compatibility.

Now you can skip to section 4 to finish setting up your conda environment.

3 Linux Installation

3.1 Hardware

The use of a dedicated GPU is strongly advised for use in any deep learning work. A CPU is adequate in most cases for making predictions with a trained model (inferencing) but the training process itself can take hours on a good GPU or many days on a plain CPU. This section of the guide assumes you are setting up a Linux workstation with a Nvidia GPU. If setting up a CPU-only workstation, skip to section 3.3.

3.2 CUDA

Full utilization of a GPU requires installing special drivers and toolkits.

3.2.1 Installing CUDA toolkit

Go to NVidia’s website.

Select Linux, x86_64, Ubuntu, 16.04, deb (local) or the correct package for your OS and download.

Change terminal directory to folder of the downloaded package.

In terminal:

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-
                        ga2_8.0.61-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

Download the patch from Nvidia’s website as well then install:

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-cublas...
sudo apt-get update
sudo apt-get upgrade cuda
sudo apt-key add /var/cuda-repo-8-0-local-ga2/7fa2af80.pub
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
```

Reboot the machine, then in terminal enter:

```
cat /proc/driver/nvidia/version
```

to ensure that the drivers installed correctly.

3.2.2 Installing cuDNN

Go to NVidia's website and login or make a free account.

Download cuDNN v6.0 for CUDA 8.0 as well as the runtime library, developer library, and code samples and user guide. The versions should match your OS, i.e. Ubuntu 16.04 (Deb).

Navigate to the download directory in terminal and enter:

```
sudo dpkg -i libcudnn6_6.0.21-1+cuda8.0_amd64.deb
sudo dpkg -i libcudnn6-dev_6.0.21-1+cuda8.0_amd64.deb
export CUDA_HOME=/usr/local/cuda-8.0
sudo apt-get install libcupti-dev
```

3.3 Anaconda

Download Miniconda for Linux, Python 3 64 bit (bash installer).

Navigate to download directory in terminal and enter:

```
sudo bash Miniconda3-latest-Linux-x86_64.sh
```

Specify /opt/miniconda3 as the install directory.

Create a .condarc file for default conda environment settings:

```
conda config --add create_default_packages python=3.5
conda config --add create_default_packages spyder
conda config --add create_default_packages numpy
conda config --add create_default_packages matplotlib
conda config --add create_default_packages hdf5
```

Then, any users on this machine can create their own conda environments with:

```
conda create -n (environment-name)
```

Replace "(environment-name)" with the name of the environment. It is recommended to name it something unique and memorable, such as jjpython35.

You can then activate this environment using

```
source activate (environment-name)
```

4 Using Anaconda

4.1 Installing Python Packages

Packages are installed through two different sources: “pip” and “conda”. The installation process is essentially the same for each. An example command looks as follows:

```
conda install `package'
```

or

```
pip install `other-package'
```

If you installed Miniconda, you wil now need to install some essential packages to get started. If you installed the full Anaconda, you can skip to section 4.1.2.

4.1.1 Essential packages

If you installed Miniconda, run the following commands after activating your conda environment. After each you will be prompted to confirm installation. Once installation is complete, run the next command. If you installed Anaconda, these will already be installed. Nothing will happen if you attempt to install a package that is already installed.

```
conda install spyder
conda install numpy
conda install hdf5
pip install matplotlib
pip install h5py
pip install pyopengl
```

4.1.2 Installing TensorFlow

Now that Miniconda users are caught up, all users should install TensorFlow inside their virtual environment. It’s just another package:

```
pip install tensorflow
```

If you have a GPU and already have CUDA installed, or once you have done the installation, you can also run:

```
pip install —upgrade tensorflow-gpu
```

(Note that the extra long – is actually two ‘-’s)

4.1.3 Installing Keras

Installing Keras is just as easy:

```
pip install keras
```

4.1.4 More Packages

As you do more work in Python, more packages can be installed just as easily. For example, nibabel can be used for reading/writing NIFTI files, and pydicom does the same for DICOM files:

```
pip install nibabel
pip install pydicom
```

4.2 Using Spyder

Spyder is a great, free Integrated Development Environment (IDE) for coding in Python. Make sure you are in your Conda environment, then enter the command

```
spyder
```

to launch Spyder. If you are used to Matlab, go to "View→Window Layouts→Matlab Layout" to switch to a layout that should feel more familiar.

Another suggestion is to switch graphics plotting from "Inline" to dedicated windows. You can leave it if you would prefer to have graphs and images you display be placed directly in the console output. If you prefer to have them popout as their own figure windows, such as in Matlab, go to "Tools→Preferences", select "IPython Console" on the left. Under the "Graphics" tab, change "Backend" to Qt5 instead of "Inline".

You are now equipped with all the necessary software to construct, train, and use a deep learning model. Continue on with this guide if you wish to dive into some coding examples. Happy (deep) learning!

5 Coding Examples

Here is an introduction to using a popular and relatively beginner-friendly API for deep learning programming: Keras (https://keras.io). This demonstration will have two parts:

1. Classification of MNIST dataset
2. Segmentation of MNIST images

The MNIST dataset consists of small images of handwritten digits 0-9 and their labels. This is a common introductory problem to deep learning since it is relatively simple and can give good results quickly. The data is also readily available.

5.1 Classification

First, we will do the traditional MNIST problem: classification. Our goal is to write a model that takes the 28x28 images in the dataset and correctly predicts their labels. We will run through the code section by section. The full code should be available on my Github repository (link) under the name “CodingDemonstration.py”, so you don’t have to copy and paste each part of it.

5.1.1 Setup and data preparation

First, we will make some imports we need for this script.

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
import numpy as np
import matplotlib.pyplot as plt
```

We set some parameters we will use for the training:

```
batch_size = 128
epochs = 2
num_classes = 10
```

Now, load the MNIST dataset. The data is shuffled and split between train and test sets. We’ll just take half of the dataset to make our computations shorter.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train[::2]
y_train = y_train[::2]
x_test = x_test[::2]
y_test = y_test[::2]
```

Data is loaded as [samples,row,col]. Since it is grayscale, it has no channel data. Keras expects chanel data, so we add a singleton dimension.

```
x_train = x_train[...,np.newaxis]
x_test = x_test[...,np.newaxis]
```

When we make our Keras model, we'll need to give it input shape, so we'll define that here. Keras just needs the row, column, and channel sizes. With Python 0-indexing, that is axes 1, 2 and 3.

```
input_shape = (x_train.shape[1], x_train.shape[2], x_train.shape[3])
# Or, more succintly
intput_shape = x_train.shape[1:4]
# Python stops before the last index of a range, so 1:4 => 1,2,3
```

We finish adjusting our images by changing them to floating point and scaling them down to the range [0,1], which is preferable for deep learning applications.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

The final touch is to convert class vectors to binary class matrices. The target MNIST data comes in the form of a class label, such as ”3”. Our convolutional model will need them to be in the form of ”one-hot” vectors, such as [0,0,0,1,0,0,0,0,0,0]. This is true of any classification model- the model predicts independent classes, not the actual number. Luckily, Keras has a built in function for the conversion.

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
# Note that this adds an axis to each of these arrays
```

Our data is now all ready for training!

5.1.2 Building a Convolutional Neural Network

Now we will build our CNN that will classify these images. Building a model in Keras is as simple as adding layers in order:

```
# Start with a sequential model
model = Sequential()
# Add the first layer- a 2D convolution. For this very first layer,
# we will need to define an input shape. The rest of the layers will
# infer the shape based on the previous layer
# We will use 16 kernels (filters) with a standard size of (3,3)
# and the standard ReLU activation function
model.add(Conv2D(16, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
# Now add a second layer, this time with more kernels
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu'))
# At this point, most standard models use what's called Max Pooling
# which sub-samples the input. Instead, we will use a strided convolution
# which accomplishes the same subsampling but is faster and generally
# more effective
model.add(Conv2D(32, kernel_size=(3,3),
                 strides=(2,2),
                 activation='relu'))
# Dropout is a common regularization technique to prevent overfitting
model.add(Dropout(0.25))
# Now, flatten what's left into a single row so that a
# multi-layer perceptron can be added to the end of the model
# for classification
```

```
model.add(Flatten())
# Keras calls the standard neural network layer "Dense"
model.add(Dense(128, activation='relu'))
# Dropout is more important in MLPs than in CNNs
model.add(Dropout(0.5))
# This is our final layer. We need to output to neurons that correspond
# to each class. We also use the softmax activation, which effectively
# calculates probabilities of each class. This is the reason that
# we needed to reformat our target data previously
model.add(Dense(num_classes, activation='softmax'))
```

Compiling the model is the final step before it is ready to train. We need to define our loss function, optimizer, and any additional metric that Keras will report as the training progresses.

Categorial cross-entropy is a standard loss for this type of classification. The ADAM optimizer is widely used with good performance on the majority of deep learning applications. Finally, we ask Keras to report the accuracy, since that we what we are really interested in.

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

5.1.3 Training and evaluation

All that's left to do is to ”fit” the model to our data! We supply our training data, our batch size and epochs that were defined earlier, we ask Keras to constantly report progress (verbose), and we supply some validation data that will be evaluated at the end of every epoch so we can keep an eye on overfitting:

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Running this for 2 epochs take about 1.5 minutes on my 7th generation i5 CPU. Results can vary.

After the training is complete, we evaluate the model again on our test data to see the results. This method will return a list of the loss value, categorical cross-entropy, and the metric we chose, accuracy.

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

which should look like

```
Test loss: 0.0625762393173
Test accuracy: 0.9804
```

Let's dislay some of our test images and see how the model does:

```
# grab the first 20 images and display them in a panel
display_ims = np.rollaxis(x_test[:10,...,0],0,2).reshape(28,10*28)
fig0 = plt.figure(0)
plt.imshow(display_ims,cmap='gray')
```

Which should look like this:

Now, get the predictions on those same images and print them out. First, get the predictions. Remember that these will come in the form of probabilities for each class for each sample.

```
probabilities = model.predict(x_test[:10])
```

Now we find the maximum probability of each sample using a Numpy function called argmax, which returns the indices where maximum occurs.

```
predictions = np.argmax(probabilities,axis=1)
# Alternatively, we can have Keras do this for us with
# the predict_classes method
predictions = model.predict_classes(x_test[:10])
```

Now print these predictions in the console and see if they match up with the images!

```
print('Class predictions are:',predictions)
```

which should look like:

```
Class predictions are:  [7 1 4 4 5 0 9 1 9 8]
```

Depending on the random initialization of your model, it likely got 9/10 or 10/10 correct. Neat!

Part 1 is now complete! We can always get better results by training for more epochs, and with the full set of training data.

5.2 Segmentation

Now, perhaps a more interesting problem in the medical imaging field: segmentation. We will again use the MNIST dataset for its availability. This is not a natural choice, so it will take a little bit of work to get the data in the format we need it. Our goal will be to segment images that have 4 digits in them. We want to be able to classify the background and each digit on a pixel-wise basis. This is called semantic segmentation, where we say whether each pixel belongs to a 0, 1, 2, 3,...

5.2.1 Data preparation

We first need to rearrnage our data into these 4-digit images.

We will make them square- 2 by 2 digits. There's many ways to do this, but the most straightforward way is a for loop.

Since we will need to apply this to both the training and testing data, let's make a function that can be used on both.

```
def ConvertXdata(x_data):
        # pre-allocate an array
        newSampNum = np.round(x_data.shape[0]/4).astype(np.int)
        x_out = np.zeros((newSampNum,2*28,2*28))
        # loop over all the data, jumping 4 at a time
        # since 4 digits are used in each sample
        for ii in range(0,x_data.shape[0],4):
                samp = np.round(ii/4).astype(np.int)
                x_out[samp,:28,:28] = x_data[ii,...,0]
                x_out[samp,28:,:28] = x_data[ii+1,...,0]
                x_out[samp,28:,28:] = x_data[ii+2,...,0]
                x_out[samp,:28,28:] = x_data[ii+3,...,0]
        return x_out[...,np.newaxis]
```

Now we can apply this function we just defined:

```
x_train2 = ConvertXdata(x_train)
x_test2 = ConvertXdata(x_test)
```

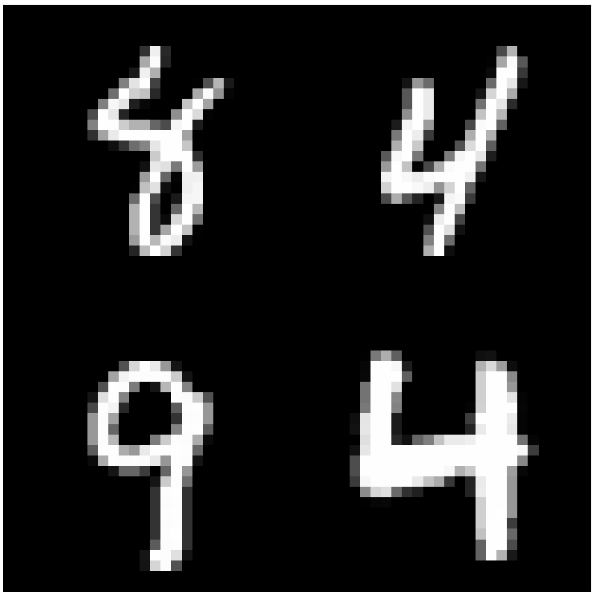
And while we're at it, let's get the input shape for setting up our model later.

```
input_shape2 = x_train2.shape[1:4]
```

Let's display one of each to make sure we did it correctly.

```
testnum = 20
fig1 = plt.figure(1)
plt.imshow(x_train2[testnum,...,0],cmap='gray')
fig2 = plt.figure(2)
plt.imshow(x_test2[testnum,...,0],cmap='gray')
```

Each of these should look something like this:



Looks great! But deep learning needs both inputs and targets... so we have to make targets for our segmentation.

Again, there are a variety of ways to go about this. A straightforward way is to first create the labeled images, then reshape them into 2x2 digits.

We'll create a function for this as well since we have both training and testing targets. Keep in mind that we now have 11 classes- the 10 different digits, and the background.

```
def ConvertYdata(x_data,y_data):
        seg = np.zeros(x_data.shape[:-1])
        for ii in range(0,seg.shape[0]):
                curim = x_data[ii,...,0]
                curim[curim>0.1] = np.argmax(y_data[ii])+1
                seg[ii,...] = curim

        newSampNum = np.round(x_data.shape[0]/4).astype(np.int)
        y_out = np.zeros((newSampNum,2*28,2*28))
        for ii in range(0,x_data.shape[0],4):
                samp = np.round(ii/4).astype(np.int)
                y_out[samp,:28,:28] = seg[ii]
                y_out[samp,28:,:28] = seg[ii+1]
                y_out[samp,28:,28:] = seg[ii+2]
                y_out[samp,:28,28:] = seg[ii+3]

        return keras.utils.to_categorical(y_out, num_classes+1)
```

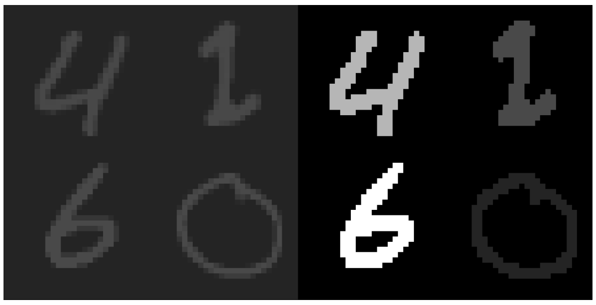
Apply the function to train and test targets.

```
y_train2 = ConvertYdata(x_train,y_train)
y_test2 = ConvertYdata(x_test,y_test)
```

It's important to make sure our inputs match up with our targets. Let's display them side by side to make sure it's right.

```
testnum = 8
fig3 = plt.figure(3)
plt.imshow(np.c_[x_train2[testnum,...,0],
                        np.argmax(y_train2[testnum],axis=2)-1],
                        cmap='gray')
fig4 = plt.figure(4)
plt.imshow(np.c_[x_test2[testnum,...,0],
                        np.argmax(y_test2[testnum],axis=2)-1],
                        cmap='gray')
```

which should look something like:



Putting your cursor over the right side of the images will show the image values in the corner of the figure. Background should be -1, while the pixels of our digits should have the correct value.

Great! Our data is now ready for training!

5.2.2 Building a segmentation network

When we did the classification scheme, we ended up with a tiny, subsampled image that was then flattened for the final neural network steps. For segmenation, we need to have classification for each pixel, so we must do something to counteract the subsampling that occurs with convolutions.

Introducing: Transposed Convolutions

```
from keras.layers import Conv2DTranspose
```

So we will use these layers to regain the spatial resolution we need for pixel-wise classification.

With that in mind, let's build our segmentation model!

We'll start the same way:

```
model2 = Sequential()
```

Now add our convoutional layers.

```
# Fewer kernels this time since our model needs to be bigger
model2.add(Conv2D(10, kernel_size=(3, 3),
                        activation='relu',
                        input_shape=input_shape2))
model2.add(Conv2D(20, kernel_size=(3,3),
                        activation='relu'))
```

Here's our strided downsampling layer. We use a 4x4 kernel so that the image sizing works out in the end.

```
model2.add(Conv2D(20, kernel_size=(4,4),
                strides=(2,2),
                activation='relu'))
model2.add(Conv2D(30, kernel_size=(3,3),
                activation='relu'))
```

Now it's time to go 'back up': regain that lost spatial resolution. Same layers in reverse order- just tranpose convolutions

```
model2.add(Conv2DTranspose(30,kernel_size=(3,3),
                        activation='relu'))
model2.add(Conv2DTranspose(30,kernel_size=(3,3),
                        strides=(2,2),
                        activation='relu'))
model2.add(Conv2DTranspose(20,kernel_size=(4,4),
                        activation='relu'))
```

Final layer: use 11 filters to correspond to our 11 classes and use softmax activation.

```
model2.add(Conv2DTranspose(11,kernel_size=(3,3),
                          activation='softmax'))
```

Let's print out a summary of the model to make sure it's what we want.

```
model2.summary()
```

The final output shape is the size of our image with the correct number of classes, 56x56x11. Perfect!

Note: "none" means not fixed. The batch size hasn't been set yet so the first dimension doesn't have a fixed size.

5.2.3 Model compiling and training

The rest of the steps are pretty much the same. Compile the model, per usual. This time we won't have the accuracy metric since that doesn't make as much sense here.

```
model2.compile(loss=keras.losses.categorical_crossentropy,
               optimizer=keras.optimizers.Adam())
```

Now run the fitting. Let's change our batch size since our inputs are 4 times larger. We'll just do 2 epochs again since this model is larger and will be a lot slower to compute.

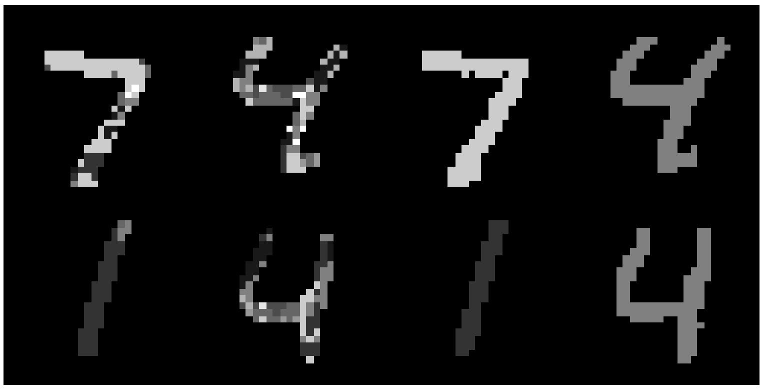
```
batch_size2 = 32
epochs2 = 2

model2.fit(x_train2, y_train2,
       batch_size=batch_size2,
       epochs=epochs2,
       verbose=1,
       validation_data=(x_test2, y_test2))
```

The loss in this case is difficult to interpret. Let's see some results!

We'll display the prediction and truth next to each other and see how it faired.

```
predictions2 = model2.predict_classes(x_test2[:2])
fig5 = plt.figure(5)
plt.imshow(np.c_[predictions2[0]-1,
               np.argmax(y_test2[0],axis=2)-1],
               cmap='gray')
```

Truth is on the right, the model's predictions are on the left.



Ooh... not great. Results may vary from simply mediocre to downright terrible. It turn out that segmenting an image 4 times larger is much more difficult than simply classifying the small images.

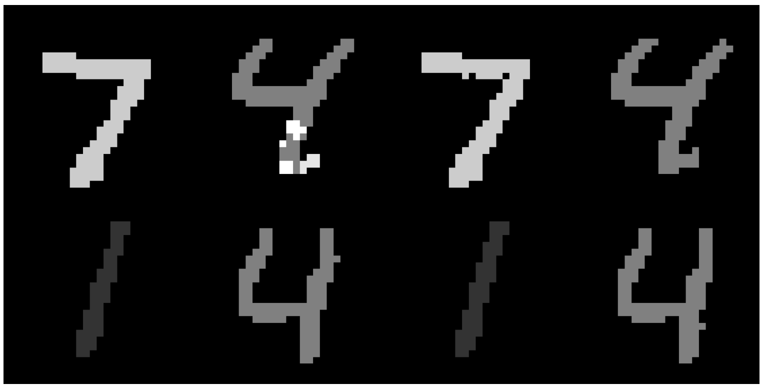
There are a variety of directions to go from here

A deeper net gives more representational power to the model. If the problem is too complex for the current network, making it deeper should improve performance.

Some mathematical tricks, like batch normalization and ELU activations can help with the learning process and make the model learn quicker.

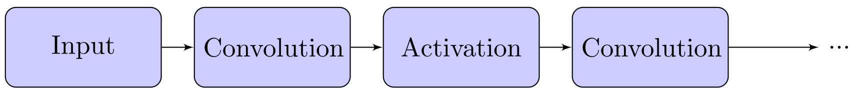
In segmentation, a particularly useful trick is the use of skip connetions, in which layers from the downsampling part of the network are concatenated with layers on the upsampling part. This both boosts the representational power of the model as well as improves the gradient flow, which also helps the model learn quicker.

However, in this case, our segmentation is still pretty simple. I ran the training for 30 epochs on a GPU and the results were much better.

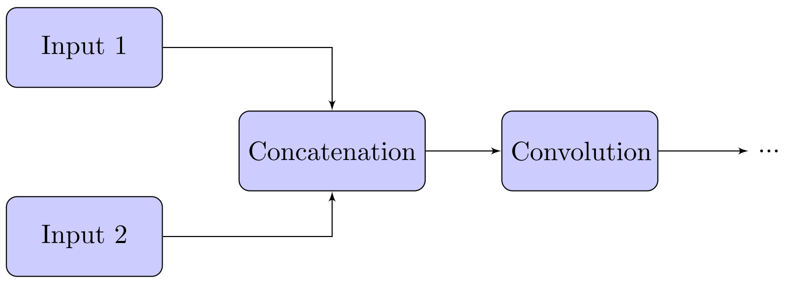


5.3 Functional API

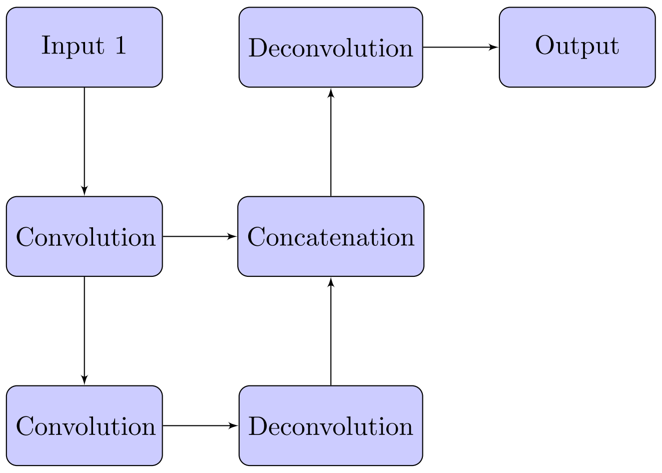
So far, we've been what's called the 'Sequential' API (Application Programming Interface), which is OK for simple models. Basically, it means that our network has a single, straight path, i.e.



Each layer has a single input and output. But what if we wanted something more complicated? What if we had two different inputs, for example? Then we would want something like



Or maybe, what I mentioned as an alteration to the segmentation network:



The extra connection shown is called a skip connection. Skip connections allow the model to consider features that were calculated earlier in the network again, merged with further processed features in practice, this has shown to be hugely helpful in geting precise localization in segmentation outputs. To make networks with this more complicated structure, we can't use the Sequential API. We need the Functional API.

There are many, many reasons to want to use the functional API. However, we will focus on the segmentation application as before, and show how a simple tweak in the functional API can lead to significantly better results.

We'll use the same segmentation data so no need to prepare anything new. Let's jump into model creation.

5.3.1 Building a Segmentation Model With Skip Connections

The functional model is just called "Model"

```
from keras.models import Model
```

Two new layers we will need for this model

```
from keras.layers import Input, concatenate
```

Creating a model in this way takes 2 arguments:

1. Inputs
2. Outputs

So, we follow a process like this:

- Define our input layer(s)
- Create the rest of our network connected to those inputs
- When we get to our final output layer(s), provide those and the input(s) to "Model", and the result will be our functional model!

Our first layer will be an "Input layer", which is where we define the input shape we will be providing during training.

```
inp = Input(shape=input_shape2)
```

Right now, 'inp' defines our input layer. In the next step, we will provide 'inp' as an argument to our next layer, which will then be called 'x1'. These variables don't matter too much. What matters is connecting the layers together in the proper order. Adding our first convolutional layer looks like this:

```
x1 = Conv2D(10,kernel_size=(3,3),activation='relu')(inp)
```

Notice that we don't need to define the input shape for this layer, since we just did that in the Input layer.

Let's build the rest of the encoding side of the network

```
x2 = Conv2D(20, kernel_size=(3,3),
                 activation='relu')(x1)

x3 = Conv2D(20, kernel_size=(2,2),
                strides=(2,2),
                activation='relu')(x2)
x4 = Conv2D(30, kernel_size=(3,3),
                 activation='relu')(x3)
```

We will use kernel size of (2,2) for the strided convolution to make matching up layers easier later. Now for the decoding side of the network, we will put the functional API to use by including skip connections.

The first layer is the same as usual. I'll add a 'd' for 'decoding'.

```
x3d = Conv2DTranspose(30,kernel_size=(3,3),
                          activation='relu')(x4)
```

Concatenate corresponding layers from the encoding and decoding sides of the network. It can be tough to get layers to match up just right in size. Playing around with kernel size and strides is usually needed so that concatenation can take place. The x,y spatial dimensions must be the same. Number of channels doesn't matter.

```
cat = concatenate([x3d,x3])
```

Now continue to add layers for the decoding side of the network, treating this merged layer like any other

```
x2d = Conv2DTranspose(30,kernel_size=(2,2),
                              strides=(2,2),
                              activation='relu')(cat)
cat = concatenate([x2d,x2])
```

Notice that we can overwrite our previously set variable 'cat' without distrupting the network. The layers are still connected in the order we set them.

```
x1d = Conv2DTranspose(20,kernel_size=(3,3),
                               activation='relu')(cat)
cat = concatenate([x1d,x1])
# Final output layer
out = Conv2DTranspose(11,kernel_size=(3,3),
                               activation='softmax')(cat)
```

All of our layers and their connections are now defined. This is commonly referred to as a 'graph', where the layers are the nodes and the calculations from layers to layers are the edges. You can think of the information 'flowing' from the inputs to the outputs. Notice how we don't give our model any real data until the training begins? The calculations are done using tensors, which are not defined arrays but rather placeholders for whichever data we feed in. Now you know how "TensorFlow" got its name!

To turn our graph into a real Keras model that we can use, simply call the 'Model' function we loaded.

```
func_model = Model(inp,out)
```

We can print out a summary of the model to make sure it's what we want. It's a little bit harder to keep track of layers in non-sequential format, but it's still a good way to make sure things look right.

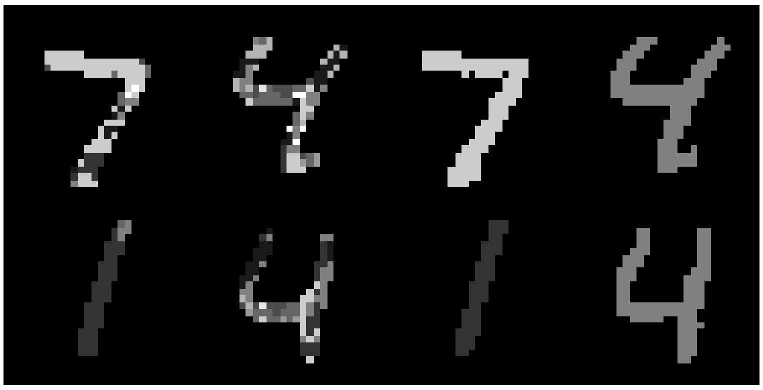
```
func_model.summary()
```

5.3.2 Testing functional model

Now, everything else is just like the previous segmentation model. Let's try it out and see how it works!

```
func_model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam())
func_model.fit(x_train2, y_train2,
          batch_size=batch_size2,
          epochs=epochs2,
          verbose=1,
          validation_data=(x_test2, y_test2))
```

```
predictionsF = model2.predict_classes(x_test2[:2])
fig6= plt.figure(6)
plt.imshow(np.c_[predictionsF[0]-1,np.argmax(y_test2[0],axis=2)-1],
                cmap='gray')
```



Well.... ok. It's about the same. However! In the long run (more than 2 epochs), having these skip connections will definitely make a difference. The difference becomes more pronounced for deeper networks (more layers) with more parameters and larger images.

Now that you know the functional API, you can make any graph you like, train it, and use it! Once you've mastered the syntax and conceptual understanding of how to connect layers, you are only limited by your imagination as far as what kind of network you can build.

Best of luck, and happy deep learning!