ADVENTURES IN MACHINE LEARNING

LEARN AND EXPLORE MACHINE LEARNING

ABOUT CONTACT

Keras tutorial – build a convolutional neural network in 11 lines

May 17, 2017 Andy Convolutional Neural Networks, Deep learning, Keras 0

Keras logo

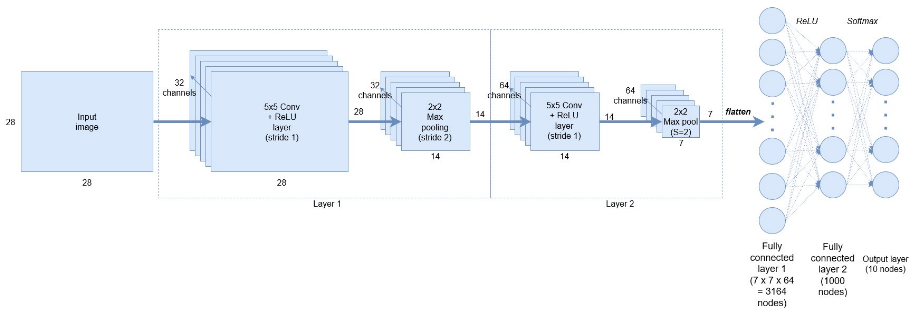
POPULAR TUTORIALS

0

In a previous tutorial, I demonstrated how to create a convolutional neural network (CNN) using TensorFlow to classify the MNIST handwritten digit dataset. TensorFlow is a brilliant tool, with lots of power and flexibility. However, for quick prototyping work it can be a bit verbose. Enter Keras and this Keras tutorial. Keras is a higher level library which operates over either TensorFlow or Theano, and is intended to stream-line the process of building deep learning networks. In fact, what was accomplished in the previous tutorial in TensorFlow in around 42 lines* can be replicated in only 11 lines* in Keras. This Keras tutorial will show you how to do this.

*excluding input data preparation and visualisation

This Keras tutorial will show you how to build a CNN to achieve >99% accuracy with the MNIST dataset. It will be precisely the same structure as that built in my previous convolutional neural network tutorial and the figure below shows the architecture of the network:


28
Input image
28
32 channels
5x5 Conv + ReLU layer (stride 1)
28
28
32 channels
2x2 Max pooling (stride 2)
14
14
64 channels
5x5 Conv + ReLU layer (stride 1)
14
14
64 channels
2x2 Max pool (S=2)
7
7
flatten
Layer 1
Layer 2
ReLU
Softmax
Fully connected layer 1 (7 x 7 x 64 = 3164 nodes)
Fully connected layer 2 (1000 nodes)
Output layer (10 nodes)

Convolutional neural network that will be built

The full code of this Keras tutorial can be found **here**.

Neural Networks Tutorial – A Pathway to Deep Learning

Python TensorFlow Tutorial – Build a Neural Network

Convolutional Neural Networks Tutorial in TensorFlow

CATEGORIES

Convolutional Neural Networks

Deep learning

Keras

Neural networks

Optimisation

TensorFlow

NEWSLETTER + FREE EBOOK

Email address:

Your email address

SIGN UP

SHARE ON SOCIAL MEDIA

0

FIND US ON FACEBOOK

Adventures in Macl

Like Page 23 likes

Be the first of your friends to like this

The main code in this Keras tutorial

The code below is the “guts” of the CNN structure that will be used in this Keras tutorial:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2,
2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

I’ll go through most of the lines in turn, explaining as we go.

```
model = Sequential()
```

Models in Keras can come in two forms – Sequential and via the Functional API. For most deep learning networks that you build, the Sequential model is likely what you will use. It allows you to easily stack sequential layers (and even recurrent layers) of the network in order from input to output. The functional API allows you to build more complicated architectures, and it won’t be covered in this tutorial.

The first line declares the model type as Sequential().

```
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu',
                 input_shape=input_shape))
```

Next, we add a 2D convolutional layer to process the 2D MNIST input images. The first argument passed to the Conv2D() layer function is the number of output channels – in this case we have 32 output channels (as per the architecture shown at the beginning). The next input is the kernel_size, which in this case we have chosen to be a 5×5 moving window, followed by the strides in the *x* and *y* directions (1, 1). Next, the activation function is a rectified linear unit and finally we have to supply the model with the size of the input to the layer (which is declared in another part of the code – see **here**). Declaring the input shape is only required of the first layer – Keras is good enough to work out the size of the tensors flowing through the model from there.

Also notice that we don't have to declare any weights or bias variables like we do in TensorFlow, Keras sorts that out for us.

```
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2,
2)))
```

Next we add a 2D max pooling layer. The definition of the layer is dead easy. We simply specify the size of the pooling in the *x* and *y* directions – (2, 2) in this case, and the strides. That's it.

```
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Next we add another convolutional + max pooling layer, with 64 output channels. The default *strides* argument in the Conv2D() function is (1, 1) in Keras, so we can leave it out. The default *strides* argument in Keras is to make it equal ot the pool size, so again, we can leave it out.

The input tensor for this layer is (batch_size, 28, 28, 32) – the 28 x 28 is the size of the image, and the 32 is the number of output channels from the previous layer. However, notice we don't have to explicitly detail what the shape of the input is – Keras will work it out for us. This allows rapid assembling of network architectures without having to worry too much about the sizes of the tensors flowing around our networks.

```
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Now that we've built our convolutional layers in this Keras tutorial, we want to flatten the output from these to enter our fully connected layers (all this is detailed in the convolutional neural network tutorial in TensorFlow). In TensorFlow, we had to figure out what the size of our output tensor from the convolutional layers was in order to flatten it, and also to determine explicitly the size of our weight and bias variables. Sure, this isn't too difficult – but it just makes our life easier not to have to think about it too much.

The next two lines declare our fully connected layers – using the Dense() layer in Keras. Again, it is very simple. First we specify the size – in line with our architecture, we specify 1000 nodes, each activated by a ReLU function. The second is our soft-max classification, or output layer, which is the size of the number of our classes (10 in this case, for our 10 possible hand-written digits).

That's it – we have successfully developed the architecture of our CNN in only 8 lines. Now let's see what we have to do to train the model and perform predictions.

Training and evaluating our convolutional neural network

We have now developed the architecture of the CNN in Keras, but we haven't specified the loss function, or told the framework what type of optimiser to use (i.e. gradient descent, Adam optimiser etc.). In Keras, this can be performed in one command:

```
model.compile(loss=keras.losses.categorical_crossentropy
,
              optimizer=keras.optimizers.SGD(lr=0.01),
              metrics=['accuracy'])
```

Keras supplies many loss functions (or you can build your own) as can be seen **here**. In this case, we will use the standard cross entropy for categorical class classification (keras.losses.categorical_crossentropy). Keras also supplies many optimisers – as can be seen **here**. In this case, we'll use the Adam optimizer (keras.optimizers.Adam) as we did in the CNN TensorFlow tutorial. Finally, we can specify a metric that will be calculated when we run evaluate() on the model. In TensorFlow we would have to define an accuracy calculating operation which we would need to call in order to assess the accuracy. In this case, Keras makes it easy for us. See **here** for a list of metrics that can be used.

Next, we want to train our model. This can be done by again running a single command in Keras:

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

This command looks similar to the syntax used in the very popular scikit learn Python machine learning library. We first pass in *all* of our training data – in this case *x_train* and *y_train*. The next argument is the batch size – we don't have to explicitly handle the batching up of our data during training in Keras, rather we just specify the batch size and it does it for us (I have a post on mini-batch gradient descent if this is unfamiliar to you). In this case we are using a batch size of 128. Next we pass the number of training epochs (10 in this case). The verbose flag, set to 1 here, specifies if you want detailed information being printed in the console about the progress of the training. During training, if verbose is set to 1, the following is output to the console:

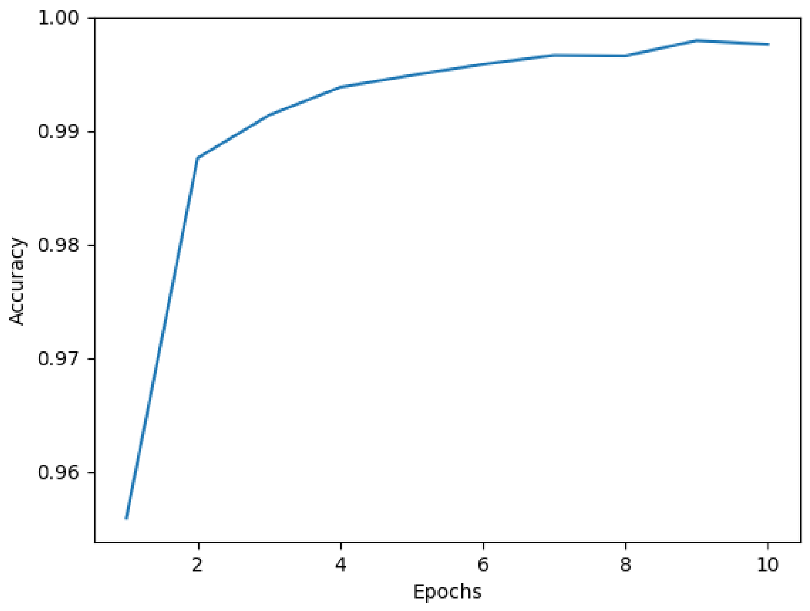
```
3328/60000 [>.............................] - ETA: 87s -
loss: 0.2180 - acc: 0.9336
3456/60000 [>.............................] - ETA: 87s -
loss: 0.2158 - acc: 0.9349
3584/60000 [>.............................] - ETA: 87s -
loss: 0.2145 - acc: 0.9350
3712/60000 [>.............................] - ETA: 86s -
loss: 0.2150 - acc: 0.9348
```

Finally, we pass the validation or test data to the fit function so Keras knows what data to test the metric against when evaluate() is run on the model. Ignore the callbacks argument for the moment – that will be discussed shortly.

Once the model is trained, we can then evaluate it and print the results:

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

After 10 epochs of training the above model, we achieve an accuracy of 99.2%, which is the same as what we achieved in TensorFlow for the same network. You can see the improvement in the accuracy for each epoch in the figure below:



Keras CNN MNIST training accuracy

Keras makes things pretty easy, don't you think? I hope this Keras tutorial has demonstrated how it can be a useful framework for rapidly prototyping deep learning solutions.

As a kind of appendix I'll show you how to keep track of the accuracy as we go through the training epochs, which enabled me to generate the graph above.

Logging metrics in Keras

Keras has a useful utility titled "callbacks" which can be utilised to track all sorts of variables during training. You can also use it to create checkpoints which saves the model at different stages in training to help you avoid work loss in case your poor overworked computer decides to crash. It is passed to the .fit() function as observed above. I'll only show you a fairly simple use case below, which logs the accuracy.

To create a callback we create an inherited class which inherits from keras.callbacks.Callback:

```
class AccuracyHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.acc = []

    def on_epoch_end(self, batch, logs={}):
        self.acc.append(logs.get('acc'))
```

The Callback super class that the code above inherits from has a number of methods that can be overridden in our callback definition such as *on_train_begin, on_epoch_end, on_batch_begin and on_batch_end.* The name of these methods are fairly self explanatory, and represent moments in the training process where we can "do stuff". In the code above, at the beginning of training we initialise a list *self.acc = []* to store our accuracy results. Using the *on_epoch_end()* method, we can extract the variable we want from the *logs,* which is a dictionary that holds, as a default, the loss and accuracy during training. We then instantiate this callback like so:

```
history = AccuracyHistory()
```

Now we can pass *history* to the .fit() function using the *callback* parameter name. Note that .fit() takes a list for the *callback* parameter, so you have to pass it *history* like this: [history]. To access the accuracy list that we created after the training is complete, you can simply call *history.acc*, which I then also plotted:

```
plt.plot(range(1,11), history.acc)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.show()
```

Hope that helps. Have fun using Keras.

0