



DEPLOY AN ML MODEL AS AN API WITH FLASK | [TOWARDS AI](#)

Deploying a Keras Model as an API Using Flask

Using the flask library to deploy a language identifier Keras model into a web app and URL based API.



Yan Gobeil

Jul 23 · 10 min read ★

Training a neural network to achieve a specific task is pretty fun and interesting, but the work doesn't stop when you are happy with the model's performance. It is useful to be able to share the model that you built with other people who may want to use it. This is important both for personal projects and for industry work.

This is why I wanted to learn how to deploy the models that I built to identify the language that a word is written in. I explained the details of the training in a [previous blog post](#), but it is not necessarily important to understand everything to follow this article.

I chose to start with what I think is the simplest way of deploying a model. I found that using the Flask library in python is pretty simple and offers many options. This allowed me to deploy my model into an API. An API is an *Application Programming Interface* that is there to make the link between different pieces of software. For example, when a developer wants to make a phone app where the user can take a picture, he doesn't have to do any hard coding to access the phone's camera. There is an API that makes the connection between the app and the camera. In the data science world, we often encounter APIs on websites, where they allow us to have access to the site's data in a simple way.

In my case, there is another very important reason to deploy the model using such an API. The user doesn't have to understand anything about deep learning to use the model. He just has to specify minimally the word to classify and the API calls the model on the background and sends only the result to the user. This makes the model widely accessible, more private and easy to modify. The API can be accessed using basically any programming language so nothing really depends on my implementation.

Everything I discuss here is used to make the API available locally so people who have the file can use it. Of course in real life, the next step is to make this API available online. There are various

ways of doing this but I didn't focus on that yet. Maybe it will be the topic of a future blog post or an update for this one.

The code for the APIs can be found in the files 'app_v1.py', 'app_v1_1.py' and 'app_v2.py' on the [GitHub repository](#) of the project.

Setup and background

For this particular project, I decided to deploy the model that I had previously trained to classify words in either English, French or Spanish. The input is simply a word and it needs to be converted into an array in order to be fed to the neural network. I already made the function 'word_to_array' for this that is simply imported. I in fact trained four different models to do this task so the API has the option of choosing which one to use to make a prediction. The models used are a simply connected network (called FF), a convolutional neural network (called CNN) and a recurrent neural network (called RNN). Each model was saved with Keras into files named 'model???.hdf5'. The output of the networks is the probability for each language.

Apart from the files mentioned above and their dependencies NumPy and Keras, the only extra package to install is Flask, which can be done easily using pip. Tensorflow itself is also necessary for a stupid reason.

I implemented a version of the API that makes a web form for the user to type in their word and get the prediction. I also made a prettier version of this app. I finally made a version that can be accessed directly from a coding environment.

Basics of Flask

The first thing to understand is how Flask works, at least in the context of this project. The first step is, of course, importing the methods necessary for the project. Then the Flask app must be defined and started.

The ‘request’ method is used to communicate with the app, the ‘jsonify’ method to convert a python dictionary to JSON format and the ‘render-templates’ is used to include an HTML template. The next step is to simply define the functions to be used by the app.

This is a simple function to load the Keras models to be used and store them in a dictionary. They are defined as global variables because the function is only called once to load the models. With the way the code is structured, having the models as output and using this output would have made the app load the models every time it was used, which is very inefficient. The last two lines are there for a future step that will be necessary to run the model because Keras has a compatibility issue when combined with Flask.

Now comes the key part of the app.

The first line defines where the following function will be posted on the web app. The path could be anything (‘/’ just means the home page) and there can be many different routes in a single app. The ‘methods’ option is used to specify what the route should expect to get as requests. The most frequent for this post are GET, which corresponds to the user getting information from the app, and POST, which is the user sending info to the app. There also exist PUT to update data and DELETE to delete data. After defining

the route comes the function that is used inside this route. The content of this depends a lot on the context and this is what I describe in detail below. Finally, the last bit of code is where the API actually runs. The models are loaded first and then the app is initiated. Loading the models outside the app is why I defined them as global variables above. Having them inside the app would lead to extra loading time every time the page is loaded.

Basic HTML

In order to understand the first version of the API, which involves webpages, it is necessary to have a basic understanding of HTML. I try to cover the very basics here and a lot of extra info can be found online, for example at <https://www.w3schools.com/html/>.

HTML is the language that web browsers use to make webpages. In HTML every object is made using tags of the form

```
<tagname option=...> text </tagname>
```

The most common tags contain an opening tag `<name>` at the beginning and a closing tag `</name>` at the end. Some of the most important ones are:

- headings `<h1>`, ..., `<h6>`: the `h1` tag is the most important one and then it goes down in size.
- paragraphs `<p>`: the main text is made of paragraphs.
- hyperlinks `<a>`: the option 'href' specifies the link and the text

between the tags is the text of the link.

- `<html>`: the whole code must be included between HTML tags.
- `<head>`: often present at the top to include metadata.
- `<body>`: the code for the webpage itself is included between body tags.
- line break `
`: this creates a break in the text to insert a new line wherever it is. It is alone so it doesn't need an ending tag.

The last type of tags that are necessary for this project is **forms**. These are normally not the most important ones but here they turn out to be very useful. The whole form is contained between `<form>` tags. Then the interacting parts of the form are written using `<input>` tags with various options. Every input must have a value for the 'name' option to identify it. The 'type' option obviously decides what type of input the tag is. The three that I used are:

- text: make a box to write text.
- radio: multiple choice selector. There must be a 'value' option for each choice to identify them and every selector in the same group must have the same 'name' option.
- submit: button with text on it. The text is defined using the 'value' option.

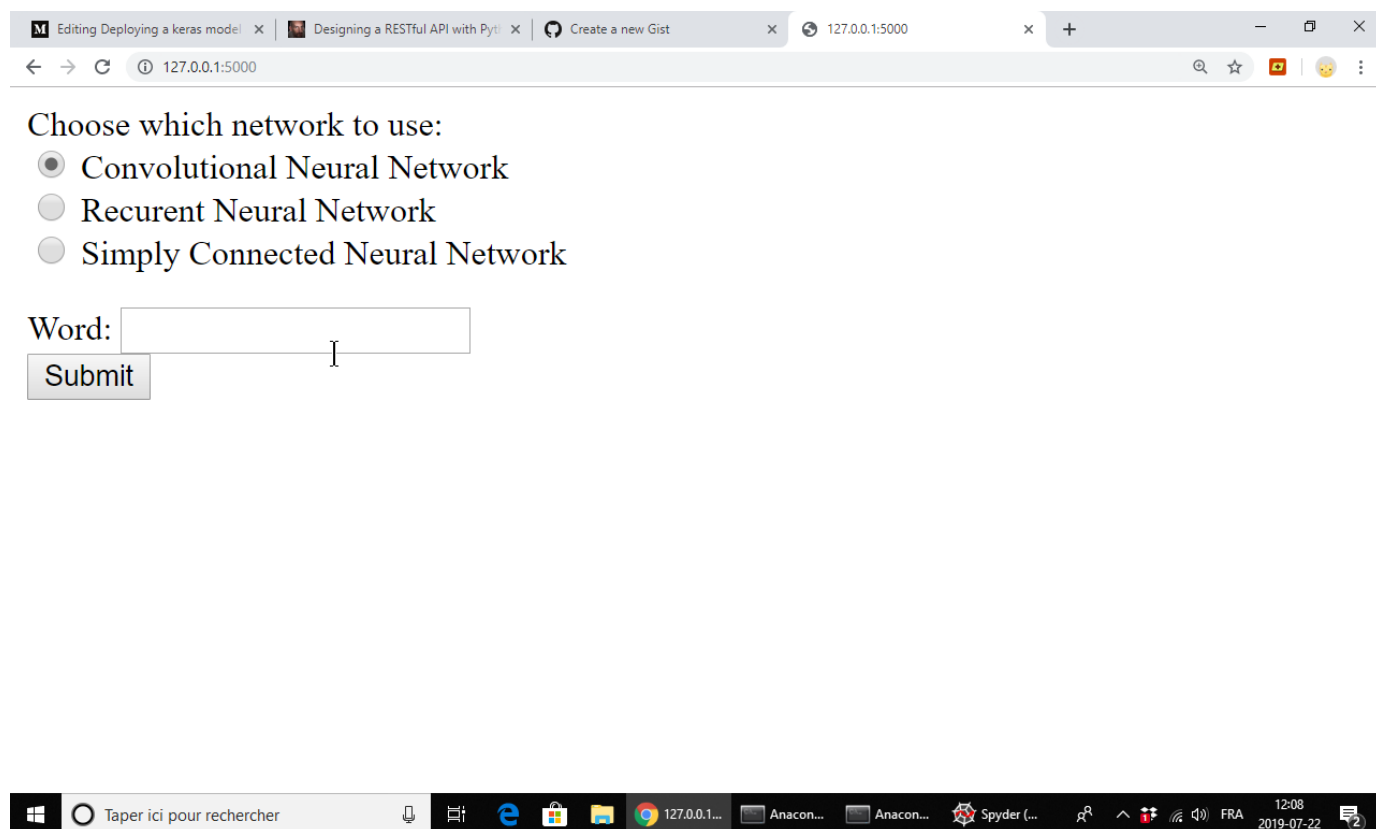
The rest of the form is plain text to fill in the gaps between the

interacting parts. Examples of all these concepts are present in the code later.

Version 1: on a webpage

The first method that I used to deploy my model was to build a simple web app that takes in a few inputs from the user, calls the Keras model and sends back the result directly in the browser. The function to use is the following.

There are a few things happening there. First, the web browser makes a GET request to obtain information from the API. This means that the first block of code is ignored since it is used only when a POST request is made, as seen from the function ‘request.method’. The function then returns the HTML code that shows the following form to the user.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:5000'. The browser tabs include 'Editing Deploying a keras model', 'Designing a RESTful API with Pyt...', 'Create a new Gist', and '127.0.0.1:5000'. The webpage content is as follows:

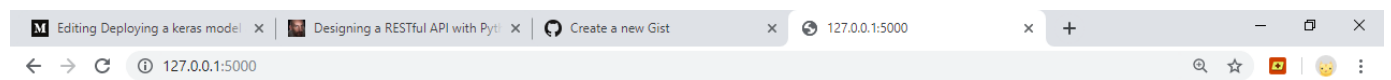
Choose which network to use:

- ☒ Convolutional Neural Network
- ☐ Recurent Neural Network
- ☐ Simply Connected Neural Network

Word:

The Windows taskbar at the bottom shows the search bar with the text 'Taper ici pour rechercher', several application icons, and the system clock displaying '12:08' and '2019-07-22'.

The 'method' option in the form tag is there to make sure that once the button is clicked a POST request is made. Then the first part of the code is executed. The 'request.form.get()' the function finds an element in the form by its name and returns its value. This is used to store the word and the name of the model to use, which is then selected from the dictionary. The next step is to convert the word to an array that will be fed to the neural network. Different models accept different shapes so there need to be conditionals there. Finally, the model is used to make a prediction. This is where the problem with Keras appears and is taken care of using a TensorFlow command. The function then returns the HTML code to display the result in the following way.



The model used is CNN

Francais: 67.36%

English: 8.07%

Espanol: 24.57%



To run this app, simply type in the command line

```
python app_v1.py
```


Some information will be displayed, including the link to use to reach the app (something like “<http://127.0.0.1:5000/>”). Copy this in a web browser and the result will be the form showed above, which you can interact with. Debug mode is off in this version because it was already tested but switching it on in the `app.run()` part gives access to a nice console that reports errors when they happen.

Version 1.1: using templates

It is impressive how HTML code for webpages can become big very fast so the previous method is not very useful for more complex pages. This is why it is possible to call premade templates in Flask and just modify a few values using the app. Bigger webpages also involve CSS (and possibly javascript) code so we need to know how to deal with that as well. The templates need to be included in a folder called “templates” and the CSS and JS are in a “static” folder. The HTML templates are just usual pages but with external variables defined between brackets `{{ }}`. The example that I used for the results page is the following.

```
<!DOCTYPE html>
<html>

<header>
  <link rel="stylesheet" href="static/style.css">
</header>

<body>
  <div class="container">
    <h1 class="word"> The model used
is {{ model }}</h1><br>
    <div class="results">
      <h1 class="result"> Francais: {{
```

```

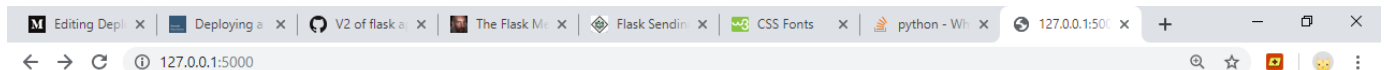
fr }}%</h1>
<h1 class="result"> English: {{ en
}}%</h1>
<h1 class="result"> Espanol: {{ es
}}%</h1>
</div>
</div>
</body>
</html>

```

The code for the home page is mostly the same as before but in a separate file. I also included some basic CSS but it is not relevant so you can see it in the style.css file on the GitHub repository. The code in the actual python script is now very simple since I just need to load the template.

The only new thing is that I used the function 'render_templates' to include the templates. I just had to give it arguments based on the names that the variables have in the HTML files.

The resulting web app looks much nicer now. (It is actually not too beautiful, but much better than before. It could be made much better with some more work.)

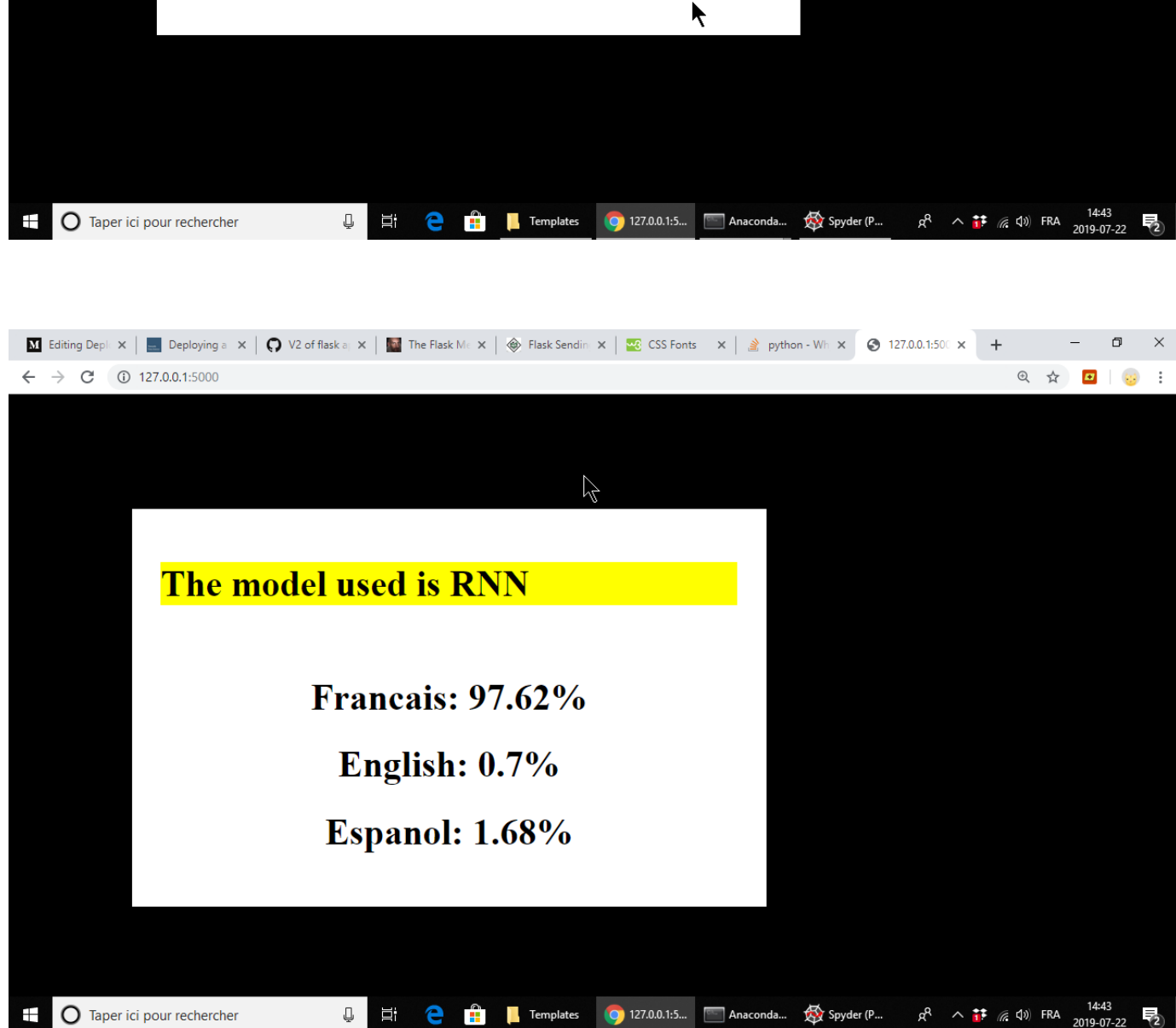


Choose which network to use:

- ☐ Convolutional Neural Network
- ☒ Recurent Neural Network
- ☐ Simply Connected Neural Network

Word:

Submit



For more complicated webpages it may be useful to know that it is possible to access the method of inputs, to do loops and define variables in a template. It is also very useful to be able to create a skeleton for the pages and inherit it for each of the webpages on the site.

Version 2: in a URL

A second way to share my model was through queries using URLs. This is much more useful for developers who would like to call the model from their own code. They would just have to use libraries like a request of urllib in python to call the API and get back the

result without ever using Keras. The function to use in this case is defined below.

This time there are no methods to specify in the app route because there is no browser involved. The user specifies the options by modifying the link that he requests and the value for the options are acquired using 'request.args.get()'. This is used here to obtain the word and the name of the model. Then the model is selected and a prediction is made. All the results are saved in a dictionary, which is transformed into a JSON format to be returned because this is what the API deals with.

To access the API just run the code as before and use the link provided to make a GET request. Here is an example using the requests library

The output of this request is a dictionary containing the requested info.

```
{'English': 28.93,  
  'Espanol': 4.5,  
  'Francais': 66.57,  
  'model': 'RNN',  
  'word': 'original'}
```

The general format for such a request for generic options is

www.api.com/page?option1=value1&option2=value2

Hopefully, this was interesting to some people. It is a less glorious part of ML/DL but it is very important for when it comes the time for people to actually use your models. I will try to write something about putting the API online. Even with this step done it is good to



Get one more story in your member preview when you sign up. It's free.



Sign up with Google



Sign up with Facebook

Already have an account? [Sign in](#)