Create and deploy Convolutional Neural networks using Keras for Image data

1. Load Data

The first step is to define the functions and classes you intend to use in this tutorial.

You will use the NumPy library to load your dataset and two classes from the Keras library to define your model.

The imports required are listed below.

```
# first neural network with keras tutorial
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
...
```

You can now load our dataset.

In this Keras tutorial, you will use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

As such, it is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values and is an ideal choice for our first neural network in Keras.

The dataset is available here:

- **Dataset CSV File (pima-indians-diabetes.csv)**
- **Dataset Details**

Download the dataset and place it in your local working directory, the same location as your Python file.

Save it with the filename:

```
pima-indians-diabetes.csv
```

Take a look inside the file; you should see rows of data like the following:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
...
```

You can now load the file as a matrix of numbers using the NumPy function **loadtxt()**.

There are eight input variables and one output variable (the last column). You will be learning a model to map rows of input variables (X) to an output variable (y), which is often summarized as *y = f(X)*.

The variables can be summarized as follows:

Input Variables (X):

1. Number of times pregnant
2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

Output Variables (y):

1. Class variable (0 or 1)

Once the CSV file is loaded into memory, you can split the columns of data into input and output variables.

The data will be stored in a 2D array where the first dimension is rows and the second dimension is columns, e.g., [rows, columns].

You can split the array into two arrays by selecting subsets of columns using the standard NumPy slice operator or ":". You can select the first eight columns from index 0 to index 7 via the slice 0:8. We can then select the output column (the 9th variable) via index 8.

```
...
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
...
```

You are now ready to define your neural network model.

Note: The dataset has nine columns, and the range 0:8 will select columns from 0 to 7, stopping before index 8. If this is new to you, then you can learn more about array slicing and ranges in this post:

- How to Index, Slice, and Reshape NumPy Arrays for Machine Learning in Python

2. Define Keras Model

Models in Keras are defined as a sequence of layers.

We create a *Sequential model* and add layers one at a time until we are happy with our network architecture.

The first thing to get right is to ensure the input layer has the correct number of input features. This can be specified when creating the first layer with the input_shape argument and setting it to (8,) for presenting the eight input variables as a vector.

How do we know the number of layers and their types?

This is a tricky question. There are heuristics that you can use, and often the best network structure is found through a process of trial and error experimentation (I explain more about this here). Generally, you need a network large enough to capture the structure of the problem.

In this example, let's use a fully-connected network structure with three layers.

Fully connected layers are defined using the Dense class. You can specify the number of neurons or nodes in the layer as the first argument and the activation function using the activation argument.

Also, you will use the rectified linear unit activation function referred to as ReLU on the first two layers and the Sigmoid function in the output layer.

It used to be the case that Sigmoid and Tanh activation functions were preferred for all layers. These days, better performance is achieved using the ReLU activation function. Using a sigmoid on the output layer ensures your network output is between 0 and 1 and is easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5.

You can piece it all together by adding each layer:

- **The model expects rows of data with 8 variables (the *input_shape=(8,)* argument).**
- **The first hidden layer has 12 nodes and uses the relu activation function.**
- **The second hidden layer has 8 nodes and uses the relu activation function.**
- **The output layer has one node and uses the sigmoid activation function.**

```
...
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
...
```

Note: The most confusing thing here is that the shape of the input to the model is defined as an argument on the first hidden layer. This means that the line of code that adds the first Dense layer is doing two things, defining the input or visible layer and the first hidden layer.

3. Compile Keras Model

Now that the model is defined, *you can compile it*.

Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU, GPU, or even distributed.

When compiling, you must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to map inputs to outputs in your dataset.

You must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network, and any optional metrics you want to collect and report during training.

In this case, use cross entropy as the loss argument. This loss is for a binary classification problems and is defined in Keras as "binary_crossentropy". You can learn more about choosing loss functions based on your problem here:

- How to Choose Loss Functions When Training Deep Learning Neural Networks

We will define the optimizer as the efficient stochastic gradient descent algorithm “adam“. This is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems. To learn more about the Adam version of stochastic gradient descent, see the post:

- Gentle Introduction to the Adam Optimization Algorithm for Deep Learning

Finally, because it is a classification problem, you will collect and report the classification accuracy defined via the metrics argument.

```
...
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
...
```

4. Fit Keras Model

You have defined your model and compiled it to get ready for efficient computation.

Now it is time to execute the model on some data.

You can train or fit your model on your loaded data by calling the fit() function on the model.

Training occurs over epochs, and each epoch is split into batches.

- Epoch: One pass through all of the rows in the training dataset
- Batch: One or more samples considered by the model within an epoch before weights are updated

One epoch comprises one or more batches, based on the chosen batch size, and the model is fit for many epochs. For more on the difference between epochs and batches, see the post:

- What is the Difference Between a Batch and an Epoch in a Neural Network?

The training process will run for a fixed number of epochs (iterations) through the dataset that you must specify using the epochs argument. You must also set the number of dataset rows that are considered before the model weights are updated within each epoch, called the batch size, and set using the batch_size argument.

This problem will run for a small number of epochs (150) and use a relatively small batch size of 10.

These configurations can be chosen experimentally by trial and error. You want to train the model enough so that it learns a good (or good enough) mapping of rows of input data to the output classification. The model will always have some error, but the amount of error will level out after some point for a given model configuration. This is called model convergence.

```
...
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
...
```

This is where the work happens on your CPU or GPU.

No GPU is required for this example, but if you're interested in how to run large models on GPU hardware cheaply in the cloud, see this post:

- How to Setup Amazon AWS EC2 GPUs to Train Keras Deep Learning Models

5. Evaluate Keras Model

You have trained our neural network on the entire dataset, and you can evaluate the performance of the network on the same dataset.

This will only give you an idea of how well you have modeled the dataset (e.g., train accuracy), but no idea of how well the algorithm might perform on new data. This

was done for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the evaluate() function and pass it the same input and output used to train the model.

This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

The evaluate() function will return a list with two values. The first will be the loss of the model on the dataset, and the second will be the accuracy of the model on the dataset. You are only interested in reporting the accuracy so ignore the loss value.

```
...
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

6. Tie It All Together

You have just seen how you can easily create your first neural network model in Keras.

Let's tie it all together into a complete code example.

```
# first neural network with keras tutorial
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

You can copy all the code into your Python file and save it as "keras_first_network.py" in the same directory as your data file "pima-indians-diabetes.csv". You can then run the Python file as a script from your command line (command prompt) as follows:

```
python keras_first_network.py
```

Running this example, you should see a message for each of the 150 epochs, printing the loss and accuracy, followed by the final evaluation of the trained model on the training dataset.

It takes about 10 seconds to execute on my workstation running on the CPU.

Ideally, you would like the loss to go to zero and the accuracy to go to 1.0 (e.g., 100%). This is not possible for any but the most trivial machine learning problems. Instead, you will always have some error in your model. The goal is to choose a model configuration and training configuration that achieve the lowest loss and highest accuracy possible for a given dataset.

```
...
768/768 [==============================] - 0s 63us/step - loss: 0.4817 - acc: 0.7708

Epoch 147/150
768/768 [==============================] - 0s 63us/step - loss: 0.4764 - acc: 0.7747

Epoch 148/150
768/768 [==============================] - 0s 63us/step - loss: 0.4737 - acc: 0.7682

Epoch 149/150
768/768 [==============================] - 0s 64us/step - loss: 0.4730 - acc: 0.7747

Epoch 150/150
768/768 [==============================] - 0s 63us/step - loss: 0.4754 - acc: 0.7799

768/768 [==============================] - 0s 38us/step

Accuracy: 76.56
```

Note: If you try running this example in an IPython or Jupyter notebook, you may get an error.

The reason is the output progress bars during training. You can easily turn these off by setting verbose=0 in the call to the fit() and evaluate() functions; for example:

```
...
# fit the keras model on the dataset without progress bars
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
```

```
# evaluate the keras model
_, accuracy = model.evaluate(X, y, verbose=0)
...
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

What score did you get?

Post your results in the comments below.

Neural networks are stochastic algorithms, meaning that the same algorithm on the same data can train a different model with different skill each time the code is run. This is a feature, not a bug. You can learn more about this in the post:

- **Embrace Randomness in Machine Learning**

The variance in the performance of the model means that to get a reasonable approximation of how well your model is performing, you may need to fit it many times and calculate the average of the accuracy scores. For more on this approach to evaluating neural networks, see the post:

- **How to Evaluate the Skill of Deep Learning Models**

For example, below are the accuracy scores from re-running the example five times:

```
Accuracy: 75.00
Accuracy: 77.73
Accuracy: 77.60
Accuracy: 78.12
Accuracy: 76.17
```

You can see that all accuracy scores are around 77%, and the average is 76.924%.

7. Make Predictions

The number one question I get asked is:

"After I train my model, how can I use it to make predictions on new data?"

Great question.

You can adapt the above example and use it to generate predictions on the training dataset, pretending it is a new dataset you have not seen before.

Making predictions is as easy as calling the predict() function on the model. You are using a sigmoid activation function on the output layer, so the predictions will be a probability in the range between 0 and 1. You can easily convert them into a crisp binary prediction for this classification task by rounding them.

For example:

```
...
# make probability predictions with the model
predictions = model.predict(X)
# round predictions
rounded = [round(x[0]) for x in predictions]
```

Alternately, you can convert the probability into 0 or 1 to predict crisp classes directly; for example:

```
...
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
```

The complete example below makes predictions for each example in the dataset, then prints the input data, predicted class, and expected class for the first five examples in the dataset.

```
# first neural network with keras make predictions
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
# summarize the first 5 cases
for i in range(5):
print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Running the example does not show the progress bar as before, as the verbose argument has been set to 0.

After the model is fit, predictions are made for all examples in the dataset, and the input rows and predicted class value for the first five examples is printed and compared to the expected class value.

You can see that most rows are correctly predicted. In fact, you can expect about 76.9% of the rows to be correctly predicted based on your estimated performance of the model in the previous section.

```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 0 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)
```

If you would like to know more about how to make predictions with Keras models, see the post:

- How to Make Predictions with Keras

Keras Tutorial Summary

In this post, you discovered how to create your first neural network model using the powerful Keras Python library for deep learning.

Specifically, you learned the six key steps in using Keras to create a neural network or deep learning model step-by-step, including:

1. How to load data
2. How to define a neural network in Keras
3. How to compile a Keras model using the efficient numerical backend
4. How to train a model on data
5. How to evaluate a model on data
6. How to make predictions with the model

Do you have any questions about Keras or about this tutorial?

Ask your question in the comments, and I will do my best to answer.

Keras Tutorial Extensions

Well done, you have successfully developed your first neural network using the Keras deep learning library in Python.

This section provides some extensions to this tutorial that you might want to explore.

- **Tune the Model. Change the configuration of the model or training process and see if you can improve the performance of the model, e.g., achieve better than 76% accuracy.**
- **Save the Model. Update the tutorial to save the model to a file, then load it later and use it to make predictions (see this tutorial).**
- **Summarize the Model. Update the tutorial to summarize the model and create a plot of model layers (see this tutorial).**
- **Separate, Train, and Test Datasets. Split the loaded dataset into a training and test set (split based on rows) and use one set to train the model and the other set to estimate the performance of the model on new data.**
- **Plot Learning Curves. The fit() function returns a history object that summarizes the loss and accuracy at the end of each epoch. Create line plots of this data, called learning curves (see this tutorial).**
- **Learn a New Dataset. Update the tutorial to use a different tabular dataset, perhaps from the UCI Machine Learning Repository.**
- **Use Functional API. Update the tutorial to use the Keras Functional API for defining the model (see this tutorial).**