In this post, we'll show how easy it is to use MLDB to build your own realtime image classification service. We will use different brand of cars in this example, but you can adapt what we show to train a model on any image dataset you want. We will be using a TensorFlow deep convolutional neural network, transfer learning, and everything will run off MLDB.

Transfer learning with the Inception model

At a high level, transfer learning allows us to take a model that was trained on one task and use its learned knowledge on another task. We use the Inception-v3 model, a deep convolutional neural network, that was trained on the ImageNet Large Visual Recognition Challenge dataset. The task of that challenge was to classify images into a varied set of 1000 classes, like badger, freight car or cheeseburger.

The Inception model was openly released as a trained TensorFlow graph. TensorFlow is a deep learning library that Google open-sourced last year, and MLDB has a built-in integration for it. As you'll see, MLDB makes it extremely simple to run TensorFlow models directly in SQL.

When solving any machine learning problem, one critical step is picking and designing feature extractors. They are used to take the thing we want to classify, be it an image, a song or a news article, and transform it into a numerical representation, called a feature vector, that can be given to a classifier. Traditionally, the selection of feature extractors was done by hand. One of the really exciting things about deep neural networks is that they can learn feature extractors themselves.

Below is the architecture of the Inception model, where images go in from the left and predictions come out to the right. The very last layer will be of size 1000 and give a probability for each of the classes. However, the layers that come before are transformations over the raw image learned by the network because they were the most useful to solve the image classification task. Some layers are for example edge detectors.

So the idea will be to run images through the network, but instead of getting the output of the last layer, that is
specialised to the ImageNet task, getting the second to last, which will give us a conceptual numerical representation of the images. We can then use that representation as features that we can give to a new classifier that we will train on our own task. So you can think of the Inception model as a way to get from an image to a feature vector over which a new classifier can efficiently operate. We are leveraging hundreds of hours of GPU compute-time that went into training the Inception model, but applying it to a completely new task.

**Inception on MLDB**

Let’s get started! The code below uses our pymldb library. You can read more about it on the [MLDB Documentation](#).

```python
print mldb.put('/v1/functions/fetch', {
  "type": 'fetcher',
  "params": {}
})

print mldb.put('/v1/functions/inception', {
  "type": 'tensorflow.graph',
  "params": {
    "modelFileUrl": 'archive+http://public.mldb.ai/models/inception_dec_2015.zip+#tensorflow_inception_graph.pb',
    "inputs": 'fetcher(url)[content] AS "DecodeJpeg/contents"',
    "outputs": "pool_3"
  }
})
```

What did we do here? We made a simple PUT call using pymldb to create the `inception` function, of type `tensorflow.graph`. It is parameterized using a JSON blob. The function loads a trained instance of the Inception model (note that MLDB can transparently load remote resources, as well as files inside of compressed archives; more on this here). We specify that the input to the model will be the remote resource located at `url`, and the output will be the `pool_3` layer of the model, which is the second to last layer. Using the `pool_3` layer will give us high level features, while the last layer called `softmax` is the one that is specialized to the ImageNet task.

Now that the `inception` function is created, it is available in SQL and as a REST endpoint. We can then run an image through the network with a simple SQL query. Here we’ll run Inception on the KDnuggets logo, and what we’ll get is the numerical representation of that image. Those 2048 numbers are what we can use as our feature vector:

```python

mldb.query("SELECT inception({url: '%s'}) as *" % kdNuggets)
```
Preparing a training dataset with SQL

Now we can import our data for training. We have a CSV file containing about 200 links to car images from 3 popular brands: Audi, BMW and Tesla. It’s important to remember that although we are using a car dataset, you could replace it with your own images of anything you want.

We can import the CSV file in a dataset by running an `import.text` procedure:

```javascript
print mldb.post("/v1/procedures", {
"type": "import.text",
"params": {
"dataFileUrl": "https://public.mldb.ai/datasets/car_brand_images/cars_urls.csv",
"outputDataset": "images"
}
})

mldb.query("SELECT * FROM images LIMIT 3")
```

We can generate some quick stats with SQL:

```javascript
mldb.query("SELECT count(*) FROM images GROUP BY brand")
```

We can now use a procedure of type `transform` to apply the `Inception` model over all images and store the results in another dataset. A transform procedure simply executes an SQL query and saves the result in a new dataset. Running the code below is essentially doing feature extraction over our image dataset.

```javascript
print mldb.post("/v1/procedures", {
"type": "transform",
"params": {
"inputData": "SELECT brand, inception({url}) as * FROM images",
"outputDataset": "training_dataset"
}
})
```

<table>
<thead>
<tr>
<th>_rowName</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;[&quot;&quot;audi&quot;&quot;][]&quot;</td>
<td>69</td>
</tr>
<tr>
<td>&quot;[&quot;&quot;bmw&quot;&quot;][]&quot;</td>
<td>72</td>
</tr>
<tr>
<td>&quot;[&quot;&quot;tesla&quot;&quot;][]&quot;</td>
<td>72</td>
</tr>
</tbody>
</table>
Training a specialized model

Now that we have features for all of our images, we use a procedure of type `classifier.experiment` to train and test a random forest classifier. The dataset will be split 50/50 between train and test by default.

Notice the contents of the `inputData` key, that specifies what data to use for training and testing, is SQL. The `{* EXCLUDING(label)}` is a good example of MLDB’s row expression syntax that is meant to work with sparse datasets with millions of columns.

```json
rez = mldb.post("/v1/procedures", {
  "type": "classifier.experiment",
  "params": {
    "experimentName": "car_brand_cls",
    "inputData": "
      SELECT
        {* EXCLUDING(brand)} as features,
        brand as label
      FROM training_dataset
      
      
    ",
    "mode": "categorical",
    "modelFileUrlPattern": "file:///mldb_data/car_brand_cls.cls",
    "configuration": {
      "type": "bagging",
      "weak_learner": {
        "type": "boosting",
        "weak_learner": {
          "type": "decision_tree",
          "max_depth": 5,
          "update_alg": "gentle",
          "random_feature_propn": 0.6
        },
        "min_iter": 5,
      }
    }
  }
})
```
"max_iter": 30
",
"num_bags": 15
}

runResults = rez.json()["status"]["firstRun"]["status"]["folds"][0]["resultsTest"]
print rez

Looking at the performance on the test set, this model is doing a pretty good job:

```
pd.DataFrame(runResults["confusionMatrix"])
    .pivot_table(index="actual", columns="predicted", fill_value=0)
pd.DataFrame.from_dict(runResults["labelStatistics"]).transpose()
```

<table>
<thead>
<tr>
<th></th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual</td>
<td></td>
</tr>
<tr>
<td>audi</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>bmw</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>tesla</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>accuracy</th>
<th>f1Score</th>
<th>precision</th>
<th>recall</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>audi</td>
<td>0.803571</td>
<td>0.633333</td>
<td>0.791667</td>
<td>0.527778</td>
<td>36</td>
</tr>
<tr>
<td>bmw</td>
<td>0.741071</td>
<td>0.623377</td>
<td>0.585366</td>
<td>0.666667</td>
<td>36</td>
</tr>
<tr>
<td>tesla</td>
<td>0.830357</td>
<td>0.781609</td>
<td>0.723404</td>
<td>0.850000</td>
<td>40</td>
</tr>
</tbody>
</table>

**Doing realtime predictions**

Now that we have a trained model, how do we use it to score new images? There are two things we need to do for this: extract the features from the image and then run that in our newly trained classifier. This is essentially our scoring pipeline.

What we do is create a function called `brand_predictor` of type `sql.expression`. This allows us to persist an SQL expression as a function that we can then call many times. When we trained our classifier above, the training procedure created a `car_brand_cls_scorer_0` automatically, available in the usual SQL/Rest, that will run the model. It will be expecting an input column named `features`. 
print mldb.put("/v1/functions/brand_predictor", {

"type": "sql.expression",

"params": {

"expression": "

car_brand_cls_scoring_0({

features: inception({url})

}) as *

"

}

})

And just like that we’re now ready to score new images off the internet:

mldb.get("/v1/functions/brand_predictor/application",

data={'input':

The image we gave it represented a Tesla, and that is the label that got the highest score.

**Conclusion**

The Machine Learning Database solves machine learning problems end-to-end, from data collection to production deployment, and offers world-class performance yielding potentially dramatic increases in ROI when compared to other machine learning platforms.

In this post, we only scratched the surface of what you can do with MLDB. We have a **white-paper** that goes over all of our design decisions in details.

If we’ve peaked your interest, here are a few links that may interest you:

Don’t hesitate to get in touch! You can find us on Gitter, or follow us on Twitter.

All the code from this article is available in the MLDB repository as a Jupyter notebook, and is also shipped with MLDB. Boot up an instance, go the the demos folder and you can run a live version.

Happy MLDBing!

**Bio:** François Maillet is a computer scientist specialising in machine learning and data science. He leads the
machine learning team at MLDB.ai, a Montréal startup building the Machine Learning Database (MLDB). François has been applying machine learning for almost 10 years to solve varied problems, like real-time bidding algorithms and behavioral modelling for the adtech industry, automatic bully detection on web forums, audio similarity and fingerprinting, steerable music recommendation and playlist generation.