Effective sampling methods within TensorFlow input functions

Will Fletcher & Laxmi Prajapat

TensorFlow World - 31st October 2019
Hello!

MSci Astrophysics
UCL, 2011-2015

Joined Datatonic
London, 2018

MChem Chemistry
Oxford, 2010-2014

Researcher & Lecturer

MSc Computational Statistics & Machine Learning
UCL, 2016-2017

Learning to walk ZOA Robotics, 2017-2018

Research Analyst esa 2014

Data Scientist Barclays, 2015-2017

MSci Astrophysics UCL, 2011-2015
About Datatonic

EXPERTISE

MACHINE LEARNING
ANALYTICS
DATA ENGINEERING

TECHNOLOGY

CLIENTS

sky
John Lewis & Partners
Transport for London
BBC
ao.com

Google Cloud
Premier Partner
Machine Learning

Google Cloud
Premier Partner
Marketing Analytics

Partner of the Year
Google Cloud

+datatonic
About the ML team

We use Google Cloud...

to accelerate Machine Learning workloads...

in a scalable way.

We work with imbalanced datasets very often...

- RETAIL
- MEDIA
- TELECOM
- FINANCE

- Fraud Detection
- Churn Prediction
- Recommender Systems
- Predictive Maintenance
We are going to talk about...

- **Theory:** Why sample?
- **Tooling:** `tf.data / tf.estimator`
- **Examples:** The simple stuff
- **Our usage:** More advanced cases
Why sample?
Dataset distributions

Frequency in **dataset**

Frequency in **model**

Frequency in **reality**

class

Frequency in dataset:

1 2 3 etc.

class

Frequency in model:

1 2 3 etc.

class

Frequency in reality:

1 2 3 etc.

class
Dataset distributions

Frequency in **dataset**

1 2 3 etc.

class

Frequency in **model**

1 2 3 etc.

class

Frequency in **reality**

1 2 3 etc.

class
Dataset distributions

Frequency in **dataset**

Frequency in **model**

Frequency in **reality**

---

**class**

1
2
3
etc.

**class**

1
2
3
etc.

**class**

1
2
3
etc.
In a classification problem, our task is to find the **boundary** between classes.
Learning from imbalanced data

The solution is **independent** of the number of examples shown (if they are informative enough)
Learning from imbalanced data

... but there are side effects

1. More examples $\Rightarrow$ more computation
   a. i/o
   b. model updates
Learning from imbalanced data

... but there are side effects

1. More examples ⇒ more computation
   a. i/o 🛠️
   b. model updates ⚙️

2. Fewer examples ⇒ poorer signal-noise ratio
   a. solution quality may suffer 🚧
   b. more variance (overfitting) likely 🎯
Learning from imbalanced data

... but there are side effects

1. More examples ⇒ more computation
   a. i/o 🔄
   b. model updates 🔄

2. Fewer examples ⇒ poorer signal–noise ratio
   a. solution quality may suffer 🚫
   b. more variance (overfitting) likely 🎯

3. Different distribution ⇒ different output probabilities
   a. will not reflect probability of future examples 🎲
Learning from imbalanced data

What is the best learning environment?
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

random initialization
Learning from imbalanced data

What is the best learning environment?

1:20 class ratio

Consider the incremental updates
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:20 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:20 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:20 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:20 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:1 class ratio

random initialization
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:1 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:1 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:1 class ratio
Learning from imbalanced data

What is the best learning environment?

Consider the incremental updates

1:1 class ratio
Learning from imbalanced data

What is the best learning environment?

More balanced batches give more information.
Learning from imbalanced data

Another technique – **example weighting**
Learning from imbalanced data

Another technique – example weighting

Number

Weighting

Contribution

\[
\text{Number} \times \text{Weighting} = \text{Contribution}
\]
Learning from imbalanced data

Another technique - example weighting
Learning from imbalanced data

Another technique – **example weighting**

The following training environments give **equivalent solutions** (with sufficient data):

<table>
<thead>
<tr>
<th>Sample balance</th>
<th>Weighting</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ratio 1:x</td>
<td>equal 1:1</td>
<td>true probability $(1+x)^{-1}$</td>
</tr>
<tr>
<td>1:1</td>
<td>1:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:x</td>
<td>x:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:1</td>
<td>1:x</td>
<td>$(1+x)^{-1}$</td>
</tr>
</tbody>
</table>
Learning from imbalanced data

Another technique – **example weighting**

The following training environments give **equivalent solutions** (with sufficient data):

<table>
<thead>
<tr>
<th>Sample balance</th>
<th>Weighting</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ratio 1:x</td>
<td>equal 1:1</td>
<td>true probability $\frac{1}{1+x}$</td>
</tr>
<tr>
<td>1:1</td>
<td>1:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:x</td>
<td>x:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:1</td>
<td>1:x</td>
<td>$\frac{1}{1+x}$</td>
</tr>
</tbody>
</table>

The easiest to learn is often given the weights $1:1$ and a threshold of $0.5$. 

---

*Note: The image contains a table and a diagram illustrating the concept of example weighting in the context of learning from imbalanced data. The table shows various sample balance and weighting scenarios along with their corresponding thresholds.*
Another technique – **example weighting**

The following training environments give **equivalent solutions** (with sufficient data):

<table>
<thead>
<tr>
<th>Sample balance</th>
<th>Weighting</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ratio 1:x</td>
<td>equal 1:1</td>
<td>true probability (1+x)^{-1}</td>
</tr>
<tr>
<td>1:1</td>
<td>1:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:x</td>
<td>x:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:1</td>
<td>1:x</td>
<td>(1+x)^{-1}</td>
</tr>
</tbody>
</table>
Learning from imbalanced data

Another technique – **example weighting**

The following training environments give equivalent solutions (with sufficient data):

<table>
<thead>
<tr>
<th>Sample balance</th>
<th>Weighting</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ratio 1:x</td>
<td>equal 1:1</td>
<td>true probability $(1+x)^{-1}$</td>
</tr>
<tr>
<td>1:1</td>
<td>1:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:x</td>
<td>x:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:1</td>
<td>1:x</td>
<td>$(1+x)^{-1}$</td>
</tr>
</tbody>
</table>

- resource-efficient (but noisier)
- easiest to learn
- calibrated
Learning from imbalanced data

Another technique – **example weighting**

The following training environments give **equivalent solutions** (with sufficient data):

<table>
<thead>
<tr>
<th>Sample balance</th>
<th>Weighting</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ratio 1:x</td>
<td>equal 1:1</td>
<td>true probability $(1+x)^{-1}$</td>
</tr>
<tr>
<td>1:1</td>
<td>1:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:x</td>
<td>x:1</td>
<td>0.5</td>
</tr>
<tr>
<td>1:1</td>
<td>1:x</td>
<td>$(1+x)^{-1}$</td>
</tr>
</tbody>
</table>

- Resource-intensive (but minimized noise)
- Resource-efficient (but noisier)
Learning from imbalanced data

Summary
Learning from imbalanced data

Summary

1. Nothing fundamentally prevents a good solution from imbalanced data
Learning from imbalanced data

Summary

1. Nothing fundamentally prevents a good solution from imbalanced data

2. Sampling or weighting to balance a dataset makes it easier to learn from
Learning from imbalanced data

Summary

1. Nothing fundamentally prevents a good solution from imbalanced data

2. **Sampling** or weighting to balance a dataset makes it easier to learn from

3. **Sampling** trades signal-noise ratio for fewer operations
Learning from imbalanced data

Summary

1. Nothing fundamentally prevents a good solution from imbalanced data

2. **Sampling** or **weighting** to balance a dataset makes it easier to learn from

3. **Sampling** trades signal-noise ratio for fewer operations

4. The probabilities output by a model reflect the distribution of data fed in
Learning from imbalanced data

Summary

1. Nothing fundamentally prevents a good solution from imbalanced data

2. **Sampling** or **weighting** to balance a dataset makes it easier to learn from

3. **Sampling** trades signal-noise ratio for fewer operations

4. The probabilities output by a model reflect the distribution of data fed in

5. **Sampling and weighting together** can give speedup without compromising interpretation of outputs as probabilities
Learning from imbalanced data

Not covered

- Cost-sensitive models / loss functions
- Data augmentation techniques e.g. SMOTE
- Imbalance-robust algorithms
But...
But... real data lives in files

This changes *everything* 🙈
A file is a **fixed** collection of examples.

**Reading** is **sequential**

**Sampling** is **random**
Sampling from files

A file is a **fixed** collection of examples

Reading is **sequential**

Sampling is **random**

easy in RAM!
Sampling from files

How do we sample the data when reading from file?

Option 1
Load all the data into memory and use `imbalanced-learn` 🐭

Option 2
Prepare a one-off random sampling of the data and save to file 🐘

Option 3
Stream the data into memory and sample on the fly 🎵🎵
Data input pipelines in TensorFlow

tf.data
tf.data

a layer between sources and inputs
We can build flexible input pipelines with the TensorFlow Dataset API (tf.data).

**Extract**
- Read from in-memory or out-of-memory datasets

**Transform**
- Apply preprocessing operations

**Load**
- Load batched examples onto the accelerator ready for processing

Methods to create a Dataset object from a data source:
- `tf.data.Dataset.from_tensor_slices`
- `tf.data.Dataset.from_generator`
- `tf.data.TFRecordDataset`
- `tf.data.TextLineDataset`

Methods to transform a Dataset:
- `tf.data.Dataset.batch`
- `tf.data.Dataset.shuffle`
- `tf.data.Dataset.map`
- `tf.data.Dataset.repeat`

Prefetch elements from the input Dataset ahead of the time they are requested by calling the `tf.data.Dataset.prefetch` method.

This transformation overlaps the work of a *producer* and *consumer*. 
Estimators (tf.estimator) is a high-level TensorFlow API that simplifies the machine learning process.

The Estimator class is an abstraction containing the necessary code to:

- run a training or evaluation loop
- predict using a trained model
- export a prediction model for use in production

The Estimator API enables us to build TensorFlow machine learning models in two ways:

**CANNED**

- “users who want to use common models”
  - Common machine learning algorithms made accessible
  - Robust with best practices encoded
  - A number of configuration options are exposed, including the ability to specify input structure using feature columns
  - Provide built-in evaluation metrics
  - Create summaries to be visualised in TensorBoard

**CUSTOM**

- “users who want to build custom machine learning models”
  - Flexibility to implement innovative algorithms
  - Fine-grained control
  - Model function (model_fn) method that build graphs for train/evaluate/predict must be written anew
  - Model can be defined in Keras and converted into an Estimator (tf.keras.estimator.model_to_estimator)
Getting data from A to B

Data for training, evaluation and prediction must be supplied through **input functions** when working with the TensorFlow Estimator API (`tf.estimator`).

```python
def input_fn():
    return features, labels
```

Read data and create Dataset object

Apply transformations

Create Iterator

In `keras.fit()`, this is done without the function wrapper.

And if you want to get batches **manually**, the final step is creating an **Iterator** object to retrieve them from the Dataset in sequence:

```python
+ tf.data.Dataset.make_one_shot_iterator
+ tf.data.Dataset.make_initializable_iterator
...
```

For example, a **one_shot_iterator** yields elements with every call of its `get_next()` method, until the Dataset is exhausted.

A valid `input_fn` takes no arguments, returning either a tuple `(features, labels)` or a Dataset generating such tuples:

- **features** – Tensor of features, or dictionary of Tensors keyed by feature name
- **labels** – Tensor of labels, or dictionary of labels keyed by label name
def make_examples(file_list):
    filenames = tf.data.Dataset.from_tensor_slices(file_list)
    filenames.shuffle(len(file_list))

    examples = filenames.interleave(
        lambda f: tf.data.TextLineDataset(f)
    )
    examples.shuffle(10**5)
    return examples
A+B ≠ B+A

The order of transformations matter.

Why?

tf.data API provides flexibility to users but the ordering of certain transformations have performance implications.

- **repeat** → **shuffle** - performant but no guarantee that samples processed in an epoch
- **shuffle** → **repeat** - guaranteed that samples processed in an epoch but less performant
Putting it all together

Key stages in the modelling pipeline:

- **Define input function** for passing data to the model for training and evaluation.
- **Define feature columns** which are specifications for how the model should interpret the input data.
- **Instantiate estimator** with necessary parameters and feeding in the feature columns.
- **Train and evaluate model.** Train loop saves model parameters as checkpoint. Eval loop restores model and uses it to evaluate model.
- **Export** trained model as SavedModel.
- **Evaluate** model - compute evaluation metrics over test data.
- **Generate predictions** with trained model.
What about TensorFlow 2.0?

Notably among the myriad of updates with the final release of TensorFlow 2.0 is the reliance on `tf.keras` as its central high-level API.

Simplified and integrated workflow for Machine Learning:

- Use `tf.data` for data loading at scale (or NumPy)
- Use `tf.keras` or existing canned estimators in `tf.estimator` for model construction

One part of the tight integration with the ecosystem is the ability for Keras models to be converted into an `Estimator` and used just like any other TensorFlow estimator.
Simple examples
TensorFlow already provides built-in functionality for sampling.

**In-memory**
- `tf.contrib.training`
  - `.resample_at_rate(inputs, rates)`
  - `.rejection_sample(tensors, accept_prob_fn, batch_size)`
  - `.stratified_sample(tensors, labels, target_probs, batch_size)`
  - `.weighted_resample(inputs, weights, overall_rate)`

**tf.data API**
- `tf.data.Dataset`
  - `.take(count)`

**tf.data.experimental**
- `.rejection_resample(class_func, target_dist)`
- `.sample_from_datasets(datasets, weights)`

**(loss sampling)**
- `tf.random`
  - `.uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max)`
  - `.log_uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max)`
  - ...
Without balancing

\[
\begin{align*}
pos &= \text{make\_examples}(\text{pos\_filenames}) \\
neg &= \text{make\_examples}(\text{neg\_filenames})
\end{align*}
\]
Without balancing

\[
\text{pos} = \text{make_examples}(\text{pos_filenames}) \\
\text{neg} = \text{make_examples}(\text{neg_filenames})
\]

\[\rightarrow \rightarrow \rightarrow \text{combine the (shuffled) datasets randomly} \]
\[\text{tf.data.experimental.sample_from_datasets}\]
Without balancing

- \( \text{pos} = \text{make_examples}(\text{pos_filenames}) \)
- \( \text{neg} = \text{make_examples}(\text{neg_filenames}) \)

- Combine the (shuffled) datasets randomly
  - `tf.data.experimental.sample_from_datasets`

- OR
  - Combine the datasets deterministically
  - `tf.data.Dataset.concatenate`
  - `tf.data.experimental.choose_from_datasets`
  - Then shuffle
With downsampling

```python
pos = make_examples(pos_filenames)
neg = make_examples(neg_filenames)
neg = neg.take(POS_SIZE)

...or

tf.data.experimental.sample_from_datasets

...and

tf.data.Dataset.concatenate
tf.data.experimental.choose_from_datasets

...then shuffle
```
With example weighting
With example weighting

```python
def pos_weighting(features, labels):
    ...

def neg_weighting(features, labels):
    ...

pos = pos.map(pos_weighting)
neg = neg.map(neg_weighting)
```

→→→ combine
With example weighting

```python
def pos_weighting(features, labels):
    weights = tf.fill(tf.shape(labels), POS_WT)
    return features, labels, weights

for tf.keras
```
def pos_weighting(features, labels):
    weights = tf.fill(tf.shape(labels), POS_WT)
    features[‘weight’] = weights
    return features, labels

for tf.estimator

estimator = DNNClassifier(
    ...
    weight_column=‘weight’
    ...
)
With example weighting

```python
def pos_weighting(features, labels):
    weights = tf.fill(tf.shape(labels), POS_WT)
    ...

def neg_weighting(features, labels):
    ...

pos = pos.map(pos_weighting)
neg = neg.map(neg_weighting)
```
Downsampling and re-weighting

```python
def pos_weighting(features, labels):
    weights = tf.fill(tf.shape(labels), POS_WT)
    ...

def neg_weighting(features, labels):
    ...

pos = pos.map(pos_weighting)
neg = neg.map(neg_weighting)
neg = neg.take(POS_SIZE)
```
Data input pipelines in TensorFlow

Summary
Data input pipelines in TensorFlow

Summary

1. `tf.data` defines ETL pipelines between data sources and model inputs
Data input pipelines in TensorFlow

Summary

1. `tf.data` defines ETL pipelines between data sources and model inputs

2. Sampling in `tf.data` is **more scalable** and enables **better practice**
Data input pipelines in TensorFlow

Summary

1. `tf.data` defines ETL pipelines between data sources and model inputs

2. Sampling in `tf.data` is **more scalable** and enables **better practice**

3. Details of pipeline design can have performance implications.
Data input pipelines in TensorFlow

Summary

1. **tf.data** defines ETL pipelines between data sources and model inputs.

2. Sampling in **tf.data** is **more scalable** and enables **better practice**.

3. Details of pipeline design can have performance implications.

4. Datasets can be iterated manually, fed to the `fit()` method of a **tf.keras** model, or returned by an `input_fn()` for the **tf.estimator** API.
Data input pipelines in TensorFlow

Summary

1. **tf.data** defines ETL pipelines between data sources and model inputs

2. Sampling in **tf.data** is more **scalable** and enables **better practice**

3. Details of pipeline design can have performance implications.

4. Datasets can be iterated manually, fed to the `fit()` method of a **tf.keras** model, or returned by an `input_fn()` for the **tf.estimator** API

5. Simple sampling can be done with **take**, **concatenate**, and **shuffle**
Data input pipelines in TensorFlow

Summary

1. `tf.data` defines ETL pipelines between data sources and model inputs

2. Sampling in `tf.data` is more scalable and enables better practice

3. Details of pipeline design can have performance implications.

4. Datasets can be iterated manually, fed to the `fit()` method of a `tf.keras` model, or returned by an `input_fn()` for the `tf.estimator` API

5. Simple sampling can be done with `take`, `concatenate`, and `shuffle`

6. More options are `sample_from_datasets` and `rejection_resample`
Where we use sampling...
Behavioural modelling

Recommender systems

Propensity to act
Examples are (user, item) pairs
Many more unobserved than observed pairs
Unobserved pairs can be generated
Behavioural modelling

Two approaches for sampling:

1. Generate all unobserved pairs on disk, and sample
   
   **Modification:** pre-sample when reading

2. Generate unobserved pairs dynamically in memory
   
   **Modification:** cache features, look up keyed by user / item
Take less, prioritise more

An effective way to handle imbalanced data is to **downsample and upweight the majority class**.

1. **Downsample** - extract random samples from the majority class known as “random majority undersampling”

2. **Upweight** - add a weighting to the downsampled examples

Weight should typically be equal to the factor used to downsample:

\[
\text{weight} = \text{original example weight} \times \text{downsampling factor}
\]

**What are the benefits?**

- **Faster convergence**: Minority class seen more often during training, helping model to converge quicker.
- **Less I/O**: Consolidating majority class into fewer examples requires less processing of data.
- **Calibration**: Upweighting ensures outputs can still be interpreted as probabilities.
Downsampling with fewer reads

No point reading the whole dataset if the model won’t read it all.
Inner workings

Create iterator

Feature copies

Weight column

features, labels

[p-01.csv, p-02.csv]

[n-01.csv, n-02.csv, n-03.csv...]

[n-10.csv, n-04.csv, n-12.csv, n-03.csv]

Dataset

multiplier = 1

Create iterator

Feature copies

Weight column

features, labels

[p-01.csv, p-02.csv]

[n-01.csv, n-02.csv, n-03.csv...]

[n-10.csv, n-04.csv, n-12.csv, n-03.csv]

Dataset

multiplier = 1
Focussing on args

`InputFnDownsampleWithWeight` is our callable class, with arguments to instantiate an `input_fn`, including the magnitude of downsampling / upweighting required.

Specify path to positive and negative examples for training and the path to examples for evaluation.

- `positive_dir` = path/to/positive/examples/*.csv
- `negative_dir` = path/to/negative/examples/*.csv
- `test_dir` = None
- `schema` = schema
- `is_train` = True
- `shuffle` = True
- `batch_size` = 128
- `num_epochs` = 5
- `header` = True
- `positive_size` = None
- `multiplier` = 1
- `weight` = 20.0
**Counting the # of examples takes O(n) time**

The number of positive examples in the dataset is calculated automatically if the value is not provided when instantiating the training input function.

This is where there is a **computational bottleneck**...

```python
def _compute_dataset_rows(dataset):
    reducer = tf.contrib.data.Reducer(
        init_func=lambda _: 0,
        reduce_func=lambda x, _: x + 1,
        finalize_func=lambda x: x)

    dataset = tf.contrib.data.reduce_dataset(dataset, reducer)
    return int(tf.Session().run(dataset))
```

**VM specification:**
- Debian GNU/Linux 9 (stretch)
- n1-standard-8
  - 8 vCPUs, 30GB memory
  - 100GB standard persistent disk
Fake example generation

No point making the fake examples on disk
Inner workings

Process as normal

unique users
random pair
multiplier = 1
repeat

unique items

sample_from_datasets

user_features.csv
item_features.csv

rejection_resample
optional, based on presence in positive dataset

tf.lookup.HashTable

Feature lookups

Feature copies

Weight column

features, labels
Speedup measurements

- **Downsampling from file**
  - Training time (seconds): 63MB pos, 5.5GB neg
  - 1 epoch: 618, 464
  - 5 epochs: 332, 246

- **Generating in memory**
  - Linear model
  - Training time (seconds): 63MB pos, 5.5GB neg
  - 1 epoch: 1983, 1068
  - 5 epochs: 513, 464

- **2-PN**
  - Training time (seconds): 63MB pos, 5.5GB neg
  - 1 epoch: 2687
  - 5 epochs: 1068
Inner workings

Process as normal

sample_from_datasets

unique users

unique items

random pair

repeat

multiplier = 1

user_features.csv

item_features.csv

tf.lookup.HashTable

Feature lookups

Feature copies

Weight column

features, labels

99:1 imbalance = 99:1 accept
i.e. 99% unnecessary overhead.
Slower than reading from file.

(n1-standard-2)
Working examples

1. Propensity Modelling - Acquire Valued Shoppers Dataset
2. Recommender Systems - Million Songs Dataset

```
ai-platform
  __init__.py
  setup.py
  trainer
    __init__.py
    task.py
ai-platform-deploy.sh
ai-platform-predict.sh
ai-platform-train.sh
local-ai-platform-train.sh
local-predict-evaluate.sh
local-train.sh
vocab
  songs.csv
  users.csv
```
Wrapping up
Final thoughts

- Other sampling methods – random replication, SMOTE etc.
- Dataset pipeline optimizations – see yesterday’s talk by Taylor Robie + Priya Gupta
- Sampling is meaningful – see “inverse probability weighting” in causal inference
TL;DL

github.com/teamdatatonic/tf-sampling

datatonic.com

@teamdatatonic

laxmi-prajapat

wjkf

TensorFlow Sampling

Effective sampling methods within TensorFlow input functions.

Table of Contents

- About the Project
  - Built With
  - Key Features
    - Sampling Techniques
    - Real-World Examples
- Getting Started
  - Installation
- Usage
  - Run Tests
- Contributing
- Licensing
Effective sampling methods within TensorFlow input functions

Laxmi Prajapat (Datatonic), William Fletcher (Datatonic)
11:50am-12:30pm Thursday, October 31, 2019
Location: Grand Ballroom H
Applications

Who is this presentation for?
- Machine learning practitioners, data scientists, and ML engineers

Level
Intermediate

Description
Many real-world machine learning applications require generative or reductive sampling of data. At training time this may be to deal with class imbalance (e.g., rarity of positives in a binary classification problem or a sparse user-item interaction matrix) or to augment the data stored on file; it may also simply be a matter of efficiency. Laxmi Prajapat and William Fletcher explore some sampling techniques in the context of recommender systems, using tools available in the tf.data API, and detail which methods are beneficial with given data and hardware demands. They present quantitative results, along with a closer examination of potential pros and cons.
Thank you for listening! 🙋‍♀️