Google Cloud

Serverless Machine Learning with Tensorflow

Vijay Reddy
Beginning of Day Logistics

- Qwiklabs Setup
  - Sign up for account
  - Provide us e-mail you signed up with
- Datalab Setup
Agenda

- **9-10:30 (90 min)**
  - Qwiklabs Setup
  - Lab 1: Explore dataset, create ML datasets, create benchmark
  - Lab 2: Getting Started with TensorFlow
- **10:30-11:00 (30 min)**
  - Break (use the break to catch up if you’re behind)
- **11:00-12:30 (90 min)**
  - Lab 3: Machine Learning using tf.estimator
  - Lab 4: Refactoring to add batching and feature-creation
- **12:30-1:30 (60 min)**
  - Lunch
- **1:30-3:00 (90 min)**
  - Lab 5: Distributed training and monitoring
  - Lab 6: Scaling up ML using Cloud ML Engine
- **3:00-3:30 (30 min)**
  - Break
- **3:30-5:00 (90 min)**
  - Lab 7: Feature Engineering
  - Wrap Up + Questions
Beginning ML Pipeline
Production Ready ML Pipeline

1. Inputs
2. Preprocessing
3. Feature Creation
4. Distributed training, Hyper-parameter tuning
5. Train model
6. Model
7. Remote Clients
8. Cloud ML
9. Prediction

REST API call with input variables
Production Ready ML Pipeline

Inputs → Preprocessing → Feature Creation → Train model → Deploy

Distributed training, Hyper-parameter tuning

Remote Clients → Prediction → Cloud ML

REST API call with input variables
Production Ready ML Pipeline

- **Inputs**
- **Preprocessing**
- **Feature Creation**
- **Distributed training, Hyper-parameter tuning**
- **Train model**
- **Model**
- **Deploy**
- **Prediction**
- **Remote Clients**
- **Cloud ML**

REST API call with input variables
Production Ready ML Pipeline

Inputs

Pre processing

Feature Creation

Train model

Distributed training, Hyper-parameter tuning

Deploy

Remote Clients

Prediction

REST API call with input variables

Cloud ML

Google Cloud
Production Ready ML Pipeline

Inputs → Preprocessing → Feature Creation → Train model → Model

Distributed training, Hyper-parameter tuning

Deploy

Remote Clients → Prediction → Cloud ML

REST API call with input variables
Production Ready ML Pipeline

Inputs → Preprocessing → Feature Creation → Train model → Model

Distributed training, Hyper-parameter tuning

Remote Clients → Prediction: load balanced, auto-scaled, scale-to-zero → Cloud ML

REST API call with input variables

Deploy: Including Model Versioning

Cloud ML
Production Ready ML Pipeline

- **Inputs**
  - SQL

- **Preprocessing**
  - Beam
  - Spark

- **Feature Creation**

- **Train model**
  - Distributed training, Hyper-parameter tuning

- **Model**
  - TensorFlow

- **Deploy**
  - On device prediction
  - Android
  - iOS

- **On device prediction**

- **Google Cloud**
Goal: To estimate taxi fare

Taxi fares:
- $2.50 initial charge
- $0.50 per ¼ mile
- 50c per minute if stopped
- Passenger pays tolls
- Various special charges

Lab 1: Explore dataset, create ML datasets, create benchmark

1. Explore
2. Create Datasets
3. Benchmark
Building Machine Learning models with TensorFlow

Data Engineering on Google Cloud Platform
Agenda

What Is TensorFlow? + Lab
TensorFlow for Machine Learning + Lab
Gaining more flexibility + Lab
Train and evaluate + Lab
Why Tensorflow?

- Community: World's most popular ML Framework
- Speed: Fast C++ engine integrates with GPUs/TPUs, support for distributed training
- Flexibility: Deploy in the cloud, on-premise, or on mobile
- Algorithms: Extensive coverage of latest deep learning algorithms
A tensor is an N-dimensional array of data

- Rank 0 Tensor: scalar
- Rank 1 Tensor: vector
- Rank 2 Tensor: matrix
- Rank 3 Tensor
- Rank 4 Tensor
TensorFlow toolkit hierarchy

- **High-level “out-of-box” API does distributed training**
  - `tf.estimator`

- **Components useful when building custom NN models**
  - `tf.layers`, `tf.losses`, `tf.metrics`

- **Python API gives you full control**
  - **Core TensorFlow (Python)**

- **C++ API is quite low-level**
  - **Core TensorFlow (C++)**

- **TF runs on different hardware**
  - CPU, GPU, TPU, Android

Run TF at scale
TensorFlow toolkit hierarchy

1. **High-level “out-of-box” API does distributed training**
   - tf.estimator

2. **Components useful when building custom NN models**
   - tf.layers, tf.losses, tf.metrics

3. **Python API gives you full control**
   - Core TensorFlow (Python)

4. **C++ API is quite low-level**
   - Core TensorFlow (C++)

5. **TF runs on different hardware**
   - CPU, GPU, TPU, Android

6. **Run TF at scale**
   - Cloud ML Engine

Source: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis non erat sem
The Python API lets you build and run Directed Graphs

```python
... c = tf.add(a, b)

session = tf.Session()
numpy_c = session.run(c, feed_dict= ....)
```
TensorFlow does lazy evaluation: you need to run the graph to get results*

```python
# numpy
a = np.array([5, 3, 8])
b = np.array([3, -1, 2])
c = np.add(a, b)
print(c)

[ 8  2 10]
```

```python
# Tensorflow

```python
a = tf.constant([5, 3, 8])
b = tf.constant([3, -1, 2])
c = tf.add(a, b)
print(c)

Tensor("Add_7:0", shape=(3,), dtype=int32)
```

```python
with tf.Session() as sess:
    result = sess.run(c)
print(result)

[ 8  2 10]
```

*TF Eager, however, allows you to execute operations imperatively*

[https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/eager](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/eager)
Graphs can be processed, compiled, remotely executed, assigned to devices

Besides computation (e.g. add, matmul), constants, variables, logging, control flow are also Ops

- biases
- weights
- examples

\[ \text{MatMul} \rightarrow \text{Add} \]

\[ \text{send} \rightarrow \text{recv} \]
TensorFlow can distribute computation
Lab 2: Getting Started with TensorFlow

In this lab you will:

- Explore the TensorFlow Python API
  - Building a graph
  - Running a graph
  - Feeding values into a graph
  - Find area of a triangle using TensorFlow
Agenda

TensorFlow for Machine Learning + Lab
TensorFlow toolkit hierarchy

High-level “out-of-box” API does distributed training

Components useful when building custom NN models

Python API gives you full control

C++ API is quite low-level

TF runs on different hardware

Run TF at scale

Cloud ML Engine

- tf.estimator
- tf.layers, tf.losses, tf.metrics
- Core TensorFlow (Python)
- Core TensorFlow (C++)
- CPU
- GPU
- TPU
- Android

Google Cloud Training and Certification
Working with Estimator API

Set up machine learning model
1. Regression or classification?
2. What is the label?
3. What are the features?

Carry out ML steps
1. Train the model
2. Evaluate the model
3. Predict with the model
The structure of an Estimator API ML model

```python
import tensorflow as tf

# Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage")
]

# Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')

# Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000])
    }  
    labels = tf.constant([50, 100])  # in thousands
    return features, labels
model.train(train_input_fn, steps=100)

# Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800])
    }
    return features
out = trained.predict(pred_input_fn)
```
import tensorflow as tf

#Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage")
]

#Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')

#Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000])
    }
    labels = tf.constant([50, 100])  # in thousands
    return features, labels
model.train(train_input_fn, steps=100)

#Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800])
    }
    return features
out = trained.predict(pred_input_fn)
Steps to do Machine Learning with model

```python
import tensorflow as tf

# Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage")
]

# Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')

# Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000])
    }
    labels = tf.constant([50, 100]) # in thousands
    return features, labels
model.train(train_input_fn, steps=100)

# Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800])
    }
    return features
out = model.predict(pred_input_fn)
```

4. Train the model
5. USE trained model to predict
Beyond linear regression with Estimators

- Deep neural network
  
  ```python
  model = DNNRegressor(feature_columns=[...],
                       hidden_units=[128, 64, 32])
  ```

- Classification
  
  ```python
  model = LinearClassifier(feature_columns=[...])
  model = DNNClassifier(feature_columns=[...], hidden_units=[...])
  ```
Lab 3: Machine Learning using tf.estimator

In this lab you will:

- Read from Pandas Dataframe into tf.constant
- Create feature columns for estimator
- Linear Regression with tf.Estimator framework
- Deep Neural Network regression
- Benchmark dataset
Agenda

Gaining more flexibility + Lab
What’s left? Ways to build effective ML

Big Data

Feature Engineering

Model Architectures
We need to refactor our Estimator model
TensorFlow toolkit hierarchy

High-level “out-of-box” API does distributed training → tf.estimator

Components useful when building custom NN models → tf.layers, tf.losses, tf.metrics

Python API gives you full control → Core TensorFlow (Python)

C++ API is quite low-level → Core TensorFlow (C++)

TF runs on different hardware → CPU, GPU, TPU, Android

Run TF at scale
To read sharded CSV files, create a TextLineDataset giving it a function to decode the CSV into features, labels

```python
CSV_COLUMNS = ['amount', 'pickuplon', 'pickuplat', 'dropofflon', 'dropofflat', 'passengers']
LABEL_COLUMN = 'amount'
DEFAULTS = [[0.0], [-74.0], [40.0], [-74.0], [40.7], [1.0]]

def read_dataset(filename, mode, batch_size=512):
    def decode_csv(value_column):
        columns = tf.decode_csv(value_column, record_defaults=DEFAULTS)
        features = dict(zip(CSV_COLUMNS, columns))
        label = features.pop(LABEL_COLUMN)
        return features, label

    dataset = tf.data.TextLineDataset(filename).map(decode_csv)

    ...
    return ...
```
Repeat the data and send it along in chunks

```python
def read_dataset(filename, mode, batch_size=512):
    ...

    dataset = tf.data.TextLineDataset(filename).map(decode_csv)
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None  # indefinitely
    else:
        num_epochs = 1  # end-of-input after this
    dataset = dataset.repeat(num_epochs).batch(batch_size)

    return dataset.make_one_shot_iterator().get_next()
```
Lab 4: Refactoring to add batching and feature-creation

In this lab you will:

- Refactor the input
- Refactor the way the features are created
- Create and train the model
- Evaluate model
Agenda

Train and evaluate + Lab
We need to make our ML pipeline more robust

In our Estimator examples so far, we:
1. ran the training_op for num_steps or num_epochs iterations
2. saved checkpoints during training
3. Used final checkpoint as model

For realistic, real-world ML models, we need to:
1. Use a fault-tolerant distributed training framework
2. Choose model based on validation dataset
3. Monitor training, especially if it will take days
4. Resume training if necessary
Estimator comes with a method that handles distributed training and evaluation

```
estimator = tf.estimator.LinearRegressor(
    model_dir=output_dir,
    feature_columns=feature_cols)
...

tf.estimator.train_and_evaluate(estimator,
    train_spec,
    eval_spec)
```

Pass in:
1. **Estimator**
2. **Train Spec**
3. **Eval Spec**

- Distribute the graph
- Share variables
- Evaluate every once in a while
- Handle machine failures
- Create checkpoint files
- Recover from failures
- Save summaries for TensorBoard
The TrainSpec consists of the things that used to be passed into the `train()` method

```python
Train_spec = tf.estimator.TrainSpec(
    input_fn=read_dataset('gs://.../train*'),
    mode=tf.contrib.learn.ModeKeys.TRAIN,
    max_steps=num_train_steps)
...

tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

Think "steps", not "epochs" with production-ready, distributed models.
1. Gradient updates from slow workers could get ignored
2. When retraining a model with fresh data, we’ll resume from earlier number of steps (and corresponding hyper-parameters)
The `EvalSpec` controls the evaluation and the checkpointing of the model since they happen at the same time.

```python
exporter = ...

eval_spec=tf.estimator.EvalSpec(
    input_fn=read_dataset('gs://.../valid*',
        mode=tf.contrib.learn.ModeKeys.EVAL),
    steps=None,
    start_delay_secs=60,  # start evaluating after N seconds
    throttle_secs=600,   # evaluate every N seconds
    exporters=exporter)

tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```
Change logging level from WARN

```python
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)
```

Log output:

```
INFO:tensorflow:Transforming feature_column _RealValuedColumn(column_name=None, dtype=tf.float32)
INFO:tensorflow:Create CheckpointSaver
INFO:tensorflow:Step 1: loss = 218.036
INFO:tensorflow:Step 101: loss = 89.9517
INFO:tensorflow:Step 201: loss = 89.9487
INFO:tensorflow:Saving checkpoints for 300 into taxi_model/model.ckpt.
INFO:tensorflow:Step 301: loss = 89.9468
INFO:tensorflow:Step 401: loss = 89.9453
INFO:tensorflow:Step 501: loss = 89.944
INFO:tensorflow:Saving checkpoints for 600 into taxi_model/model.ckpt.
INFO:tensorflow:Step 601: loss = 89.9429
INFO:tensorflow:Step 701: loss = 89.9419
INFO:tensorflow:Step 801: loss = 89.941
INFO:tensorflow:Saving checkpoints for 900 into taxi_model/model.ckpt.
INFO:tensorflow:Step 901: loss = 89.9402
```
Use TensorBoard to monitor training

Loss
Data
Model
Lab 5: Distributed training and monitoring

In this lab you will:

● Create features out of input data
● Train and evaluate
● Monitor with Tensorboard
Scaling TF models with Cloud ML Engine

Data Engineering on Google Cloud Platform
What’s left? Ways to build effective ML

Big Data

Feature Engineering

Model Architectures
Agenda

Why Cloud ML Engine?

Packing up a TensorFlow model + Lab
TensorFlow toolkit hierarchy

High-level “out-of-box” API does distributed training
- \texttt{tf.\textcolor{green}{estimator}}

Components useful when building custom NN models
- \texttt{tf.\textcolor{yellow}{layers}, tf.\textcolor{yellow}{losses}, tf.\textcolor{yellow}{metrics}}

Python API gives you full control
- Core TensorFlow (Python)

C++ API is quite low-level
- Core TensorFlow (C++)

TF runs on different hardware
- CPU, GPU, TPU, Android

Run TF at scale

Cloud ML Engine
In Datalab, start locally on sampled dataset
Then, scale it out to GCP using serverless technology.
Agenda

Packing up a TensorFlow model + Lab
Training your model with Cloud ML Engine

Step 1
Use TensorFlow to create computation graph and training application

Step 2
Package your trainer application

Step 3
Configure and start a Cloud ML Engine job
Create `task.py` to parse command-line parameters and send along to `train_and_evaluate`

**model.py**

```python
def train_and_evaluate(args):
    estimator = tf.estimator.DNNRegressor(
        model_dir=args['output_dir'],
        feature_columns=feature_cols,
        hidden_units=args['hidden_units'])
    train_spec=tf.estimator.TrainSpec(
        input_fn=read_dataset(args['train_data_paths'],
                               batch_size=args['train_batch_size'],
                               mode=tf.contrib.learn.ModeKeys.TRAIN),
        max_steps=args['train_steps'])
    exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)
    eval_spec=tf.estimator.EvalSpec(...)
    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

**task.py**

```python
parser.add_argument('--train_data_paths', required=True)
parser.add_argument('--train_steps', ...
```

Google Cloud

Training and Certification
The `model.py` contains the ML model in TensorFlow (Estimator API)

<table>
<thead>
<tr>
<th>Example of the code in model.py (see previous module)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training and evaluation input functions</strong></td>
</tr>
<tr>
<td>CSV_COLUMNS = ...</td>
</tr>
<tr>
<td>def read_dataset(filename, mode, batch_size=512):</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td><strong>Feature columns</strong></td>
</tr>
<tr>
<td>INPUT_COLUMNS = [</td>
</tr>
<tr>
<td>tf.feature_column.numeric_column('pickuplon'),</td>
</tr>
<tr>
<td><strong>Feature engineering</strong></td>
</tr>
<tr>
<td>def add_more_features(feats):</td>
</tr>
<tr>
<td># will be covered in next chapter; for now, just a no-op</td>
</tr>
<tr>
<td>return feats</td>
</tr>
<tr>
<td><strong>Serving input function</strong></td>
</tr>
<tr>
<td>def serving_input_fn():</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>return tf.estimator.export.ServingInputReceiver(features, feature_pholders)</td>
</tr>
<tr>
<td><strong>Train and evaluate loop</strong></td>
</tr>
<tr>
<td>def train_and_evaluate(args):</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)</td>
</tr>
</tbody>
</table>
Package up TensorFlow model as Python package

```text
- taxifare/
- taxifare/ PKG-INFO
- taxifare/setup.cfg
- taxifare/setup.py
- taxifare/trainer/
- taxifare/trainer/__init__.py
- taxifare/trainer/task.py
- taxifare/trainer/model.py
```

Python packages need to contain an `__init__.py` in every folder.
Verify that the model works as a Python package

```bash
export PYTHONPATH=${PYTHONPATH}:/somedir/taxifare
python -m trainer.task \
  --train_data_paths="/somedir/datasets/*train*" \
  --eval_data_paths=/somedir/datasets/*valid* \
  --output_dir=/somedir/output \
  --train_steps=100 --job-dir=/tmp
```
Then use the `gcloud` command to submit the training job, either locally or to cloud

```bash
gcloud ml-engine local train \ --module-name=trainer.task \ --package-path=/somedir/taxifare/trainer \ -- \ --train_data_paths etc.
REST as before
```

```bash
gcloud ml-engine jobs submit training $JOBNAME \ --region=$REGION \ --module-name=trainer.task \ --job-dir=$OUTDIR --staging-bucket=gs://$BUCKET \ --scale-tier=BASIC \ REST as before
```
Cloud ML makes deploying models and scaling the prediction infrastructure easy.
We can not reuse the training input function for serving
The `serving_input_fn` specifies what the caller of the `predict()` method will have to provide.

```python
def serving_input_fn():
    feature_placeholders = {
        'pickuplon': tf.placeholder(tf.float32, [None]),
        'pickuplat': tf.placeholder(tf.float32, [None]),
        'dropofflat': tf.placeholder(tf.float32, [None]),
        'dropofflon': tf.placeholder(tf.float32, [None]),
        'passengers': tf.placeholder(tf.float32, [None]),
    }
    features = {
        key: tf.expand_dims(tensor, -1)
        for key, tensor in feature_placeholders.items()
    }
    return tf.estimator.export.ServingInputReceiver(features, feature_placeholders)
```
2. Deploy trained model to GCP

```bash
MODEL_NAME="taxifare"  
MODEL_VERSION="v1"  
MODEL_LOCATION="gs://${BUCKET}/taxifare/smallinput/taxi_trained/export/Servo/.. ./"

gcloud ml-engine models create ${MODEL_NAME} --regions $REGION

gcloud ml-engine versions create ${MODEL_VERSION} --model ${MODEL_NAME}  
--origin ${MODEL_LOCATION}

Could also be a locally-trained model
```
3. Client code can make REST calls

```python
credentials = GoogleCredentials.get_application_default()
api = discovery.build('ml', 'v1', credentials=credentials,
discoveryServiceUrl='https://storage.googleapis.com/cloud-ml/discovery/ml_v1beta1_discovery.json')
request_data = [
    {'pickup_longitude': -73.885262,
     'pickup_latitude': 40.773008,
     'dropoff_longitude': -73.987232,
     'dropoff_latitude': 40.732403,
     'passenger_count': 2}]
parent = 'projects/%s/models/%s/versions/%s' % ('cloud-training-demos',
    'taxifare', 'v1')
response = api.projects().predict(body={'instances': request_data},
    name=parent).execute()
```
Lab 6: Scaling up ML using Cloud ML Engine

In this lab you will:

1. Package up the code
2. Find absolute paths to data
3. Run the Python module from the command line
4. Run locally using gcloud
5. Submit training job using gcloud
6. Deploy model
7. Prediction
8. Train on a larger dataset
9. 1-million row dataset
Improving ML through feature engineering

Data Engineering on Google Cloud Platform
Agenda

ML abstraction levels + Lab

Hyperparameter tuning + Demo

ML Abstraction Levels
What’s left? Ways to build effective ML

Big Data

Feature Engineering

Model Architectures
Raw data are converted to numeric features in different ways

In the estimator API, this is a feature column
Numeric values can be used as-is

```json
{
    "transactionId": 42,
    "name": "Ice Cream",
    "price": 2.50,
    "tags": ["cold", "dessert"],
    "servedBy": {
        "employeeId": 72365,
        "waitTime": 1.4,
        "customerRating": 4
    },
    "storeLocation": {
        "latitude": 35.3,
        "longitude": -98.7
    }
}

[ , 2.50, ..., 1.4, ]
...

INPUT_COLUMNS = [
    ..., 
    tf.feature_column.numeric_column('price'),
    ...
]

NUMERIC_COLUMN IS A TYPE OF FEATURE COLUMN
```
Overly specific attributes should be discarded

```
{
    "transactionId": 42,
    "name": "Ice Cream",
    "price": 2.50,
    "tags": ["cold", "dessert"],
    "servedBy": {
        "employeeId": 72365,
        "waitTime": 1.4,
        "customerRating": 4
    },
    "storeLocation": {
        "latitude": 35.3,
        "longitude": -98.7
    }
}
```

Don't train on IDs or other super-specific information.
Categorical values could be one-hot encoded

```python
{  
  "transactionId": 42,
  "name": "Ice Cream",
  "price": 2.50,
  "tags": ["cold", "dessert"],
  "servedBy": {
    "employeeId": "72365",
    "waitTime": 1.4,
    "customerRating": 4
  },
  "storeLocation": {
    "latitude": 35.3,
    "longitude": -98.7
  }
},

tf.feature_column.categorical_column_with_vocabulary_list('employeeId',
  keys=['8345', '72365', '87654', '98723', '23451']),
```

CATEGORICAL_COLUMN_WITH_VOCABULARY_LIST IS A TYPE OF FEATURE_COLUMN
Options for encoding categorical data

These are all different ways to create a categorical column

If you know the keys beforehand:
```
import tensorflow as tf

tf.feature_column.categorical_column_with_vocabulary_list('employeeId',
    vocabulary_list = ['8345', '72345', '87654', '98723', '23451']),
```

If your data is already indexed; i.e., has integers in [0-N):
```
import tensorflow as tf

tf.feature_column.categorical_column_with_identity('employeeId',
    num_buckets = 5)
```

If you don’t have a vocabulary of all possible values:
```
import tensorflow as tf

tf.feature_column.categorical_column_with_hash_bucket('employeeId',
    hash_bucket_size = 500)
```
Creating feature crosses using TensorFlow

```python
day_hr = tf.feature_column.crossed_column([dayofweek, hourofday], 24*7)
```
How to deal with latitudes and longitudes?

- **LatitudeBin1**: $32 < \text{latitude} \leq 33$
- **LatitudeBin7**: $37 < \text{latitude} \leq 38$
Feature crosses can simplify learning

![Diagram showing feature crosses in ML models for taxi classification.](image-url)
Creating bucketized features using TensorFlow

```python
latbuckets = np.linspace(32.0, 42.0, nbuckets).tolist()
discrete_lat = tf.feature_column.bucketized_column(lat, latbuckets)
```
What’s left? Ways to build effective ML

Big Data

Feature Engineering

Model Architectures
Animal Classifier: Can Fly or Can’t Fly?
Seagulls can fly.
Pigeons can fly.
Animals with wings can fly.
Penguins...
Well, at least they try.
Memorization + Generalization

**Wide**
Memorization: “Seagulls can fly.” “Pigeons can fly.”

**Deep**
Generalization: “*Animals with wings* can fly.”

**Wide + Deep**
Generalization + memorizing exceptions:
“*Animals with wings can fly, but penguins cannot fly.*”
Wide & Deep Mission

“Combine the power of memorization and generalization on one unified machine learning platform for everyone.”
Wide-and-deep network in tf.estimator

```python
model = tf.estimator.DNNLinearCombinedClassifier(
    model_dir=..., 
    linear_feature_columns=wide_columns, 
    dnn_feature_columns=deep_columns, 
    dnn_hidden_units=[100, 50])
```
Two types of features: Dense & sparse

```json
{
    "transactionId": 42,
    "name": "Ice Cream",
    "price": 2.50,
    "tags": ["cold", "dessert"],
    "servedBy": {
        "employeeId": 72365,
        "waitTime": 1.4,
        "customerRating": 4
    },
    "storeLocation": {
        "latitude": 35.3,
        "longitude": -98.7
    }
}
```
Linear for sparse, independent features

<table>
<thead>
<tr>
<th>0.</th>
<th>1.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
<th>0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>1.</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
</tr>
</tbody>
</table>
DNNs good for dense, highly correlated pixel_values

1024^2 input nodes
10 hidden nodes -> 10 image features
Lab 7: Feature Engineering

In this lab you will incorporate feature engineering into the pipeline.

1. Working with feature columns
2. Adding feature crosses in TensorFlow
3. Reading data from BigQuery
4. Creating datasets using Dataflow
5. Using a wide-and-deep model
Agenda

Hyperparameter tuning
Very sensitive to batch_size and learning_rate

There are a variety of model parameters too

- Size of model
- Number of hash buckets
- Embedding size
- Etc.

Wouldn’t it be nice to have the NN training loop do meta-training across all these parameters?
Cloud MLE supports hyperparameter tuning

1. Make the parameter a command-line argument
2. Make sure outputs don’t clobber each other
3. Supply hyperparameters to training job
1. Make parameter a command-line argument and use it in model

```python
parser.add_argument(  
    '--nbuckets',  
    help='Number of buckets into which to discretize lats and lons',  
    default=10,  
    type=int  
)
parser.add_argument(  
    '--hidden_units',  
    help='List of hidden layer sizes to use for DNN feature columns',  
    default="128 32 4"  
)
```
2. Make sure that outputs don’t clobber each other

```python
output_dir = os.path.join(
    output_dir,
    json.loads(
        os.environ.get('TF_CONFIG', '{}').
    ).get('task', {}).
    get('trial', '')
)
```
3. Supply hyperparameters to training

```yaml
%writefile hyperparam.yaml
trainingInput:
  scaleTier: STANDARD_1
hyperparameters:
  goal: MINIMIZE
  maxTrials: 30
  maxParallelTrials: 1
  hyperparameterMetricTag: rmse
params:
  - parameterName: train_batch_size
type: INTEGER
  minValue: 64
  maxValue: 512
  scaleType: UNIT_LOG_SCALE
  - parameterName: nbuckets
    type: INTEGER
    minValue: 10
    maxValue: 20
    scaleType: UNIT_LINEAR_SCALE
  - parameterName: hidden_units
    type: CATEGORICAL
categoricalValues: ["128 64 32", "256 128 16", "512 128 64"]
```

gcloud ml-engine jobs submit training $JOBNAME
   \--region=$REGION \--module-name=trainer.task \...
   --config=hyperparam.yaml
   -- \--output_dir=$OUTDIR \--train_steps=1000
Demo: Hyperparameter tuning

- This notebook demonstrates model & training modifications to support hyper parameter tuning

Accuracy improves through feature engineering, hyperparameter tuning, and lots of data
Production Ready ML Pipeline

Inputs

Pre processing

Feature Creation

Train model

Distributed training, Hyper-parameter tuning

Model

Deploy: Including Model Versioning

Prediction: load balanced, autocaled, scale-to-zero

Remote Clients

REST API call with input variables

Cloud ML