# 11-695: AI Engineering
# Customizing NNs

LTI/SCS

Spring 2020

# Outline

CarnegieMellon


1. Input
2. Auto Differentiation
3. Choosing Losses
4. Choosing Metrics

# Tensorizing data with `tf.data.Dataset`[1]

Carnegie Mellon

- Create a Dataset object from:
  - numpy data (you have done parsing the data into Numpy)
  - file paths to the data (you have just only done collecting paths)
- In a multiple ways:
  - `tf.data.Dataset.from_tensors`
  - `tf.data.Dataset.from_tensor_slices`
  - `tf.data.Dataset.from_generator` - memory efficiency

[1] https://www.tensorflow.org/guide/datasets

# Tensorizing data with `tf.data.Dataset`[1]

CarnegieMellon

- Create a Dataset object from:
  - numpy data (you have done parsing the data into Numpy)
  - file paths to the data (you have just only done collecting paths)
- In a multiple ways:
  - `tf.data.Dataset.from_tensors`
  - `tf.data.Dataset.from_tensor_slices`
  - `tf.data.Dataset.from_generator` - memory efficiency
- If you are sure about data format, use a child class:
  - `tf.data.TextLineDataset`
  - `tf.data.TFRecordDataset`
  - Many more experimentals: `Random`, `CSV`, `SQL`, `Kafka`, `SequenceFile`, `Kinesis`, `MachineFiles`, `LMDB`

[1] https://www.tensorflow.org/guide/datasets

# Batching with `tf.data.Dataset`



- `tf.data.Dataset.shuffle(buffer_size)`
- `tf.data.Dataset.batch(batch_size)`
- `tf.data.Dataset.repeat(n_epochs)`

# Looping `tf.data.Dataset` objects

Carnegie Mellon

tf_data.py

```python
data = tf.data.Dataset.from_tensor_slices((img_path, img_labels))
# doing shuffling, batching, repeating

# then
for batch_data in data:
  (img, labels) = batch_data
  # do smth

# or
for (batch_num, (img, labels)) in enumerate(data):
  # do smth
```

- Iterate the Same as when you work with Python

# Ingesting paths with `tf.data.Dataset`

Carnegie Mellon

tf_custom_data

```
1  def parse_data(img_path, label_path):
2    # do something to convert paths to data and and labels
3
4  def build_dataset():
5    dataset = tf.data.Dataset.from_tensor_slices((img_paths_tensor, label_paths_tensor))
6    dataset = dataset.map(parse_data,  num_parallel_calls=10)
7
8  # then
9  for batch_data in dataset:
10   (img, labels) = batch_data
11   # do smth
12
13 # or
14 for (batch_num, (img, labels)) in enumerate(dataset):
15   # do smth
```

- It is flexible to define your own way of tensorizing your data

# Fast prototyping with built-in datasets

Carnegie Mellon

- Child classes of `tf.keras.datasets`
- Modules:
  - `boston_housing`: Boston housing price regression dataset.
  - `mnist`: MNIST handwritten digits dataset.
  - `cifar10`: CIFAR10 small images classification dataset.
  - `cifar100`: CIFAR100 small images classification dataset.
  - `fashion_mnist`: Fashion-MNIST dataset.
  - `imdb`: IMDB sentiment classification dataset.
  - `reuters`: Reuters topic classification dataset.

# Fast prototyping with built-in datasets

Carnegie Mellon

tf_keras_datasets

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Add a channels dimension
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

train_ds = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)).shuffle(10000).batch(32)

test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)

# train
for images, labels in train_ds:
    train_step(images, labels)

# test
for test_images, test_labels in test_ds:
    test_step(test_images, test_labels)
```

# `tf.data` resources



- Tutorial: `tf.data`

# Outline

CarnegieMellon

1. Input
2. Auto Differentiation
3. Choosing Losses
4. Choosing Metrics

# Calculating gradients: `tf.GradientTape`

Carnegie Mellon

tf_gtape

```
x = tf.Variable([[1.0, 2.0]]) # float
with tf.GradientTape() as tape:
  tape.watch(x) # x must be a Tensor first, it's auto done
  # create 'graph' of ops here
  y = tf.square(x)

# call gradient() outside the context
# <tf.Tensor: id=92, shape=(1, 2), dtype=float32, numpy=array([[2., 4.]], dtype=float32)>
dy_x = tape.gradient(y, x)

dy_x = tape.gradient(y, x) # error! tape is not persistent
```

- Tape is a sequential storage device
- Gradient Tape records gradient in order of operations
  - Under its umbrella of *context*
  - It records ops so that later can calculate grads
  - It costs memory

# Backproping gradients: `tf.GradientTape`

Carnegie Mellon

**tf_gtape_fn**

```
1  def grad(model, x, y):
2    with tf.GradientTape() as tape:
3      loss_value = loss(model, x, y)
4    return tape.gradient(loss_value, model.variables)
5
6  # take grads
7  grads = grad(model, x, y)
8
9  # Backprop: Apply the grads to the model's variables
10 optimizer.apply_gradients(zip(grads, model.variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`
- Normally, avoid grads *calculation* inside tape's *context*
- Control to-watch variables: `watch_accessed_variables=False`

# Co-optimizing: tf.GradientTape

tf_gtape_fn_2

```
1  def Model1(tf.keras.Model):
2      ...
3
4  def Model2(tf.keras.Model):
5     ...
6
7  model1 = Model1()
8  model2 = Model2()
9
10 variables = model1.variables() + model2.variables()
11 grads = tape.gradient(loss, variables)
12
13 optimizer.apply_gradients(zip(gradients, variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`

# Co-optimizing: tf.GradientTape



tf_gtape_fn_2

```
1  def Model1(tf.keras.Model):
2      ...
3
4  def Model2(tf.keras.Model):
5     ...
6
7  model1 = Model1()
8  model2 = Model2()
9
10 variables = model1.variables() + model2.variables()
11 grads = tape.gradient(loss, variables)
12
13 optimizer.apply_gradients(zip(gradients, variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`
- Normally, avoid grads *calculation* inside tape's *context*

# Co-optimizing: `tf.GradientTape`



tf_gtape_fn_2

```
def Model1(tf.keras.Model):
    ...

def Model2(tf.keras.Model):
   ...

model1 = Model1()
model2 = Model2()

variables = model1.variables() + model2.variables()
grads = tape.gradient(loss, variables)

optimizer.apply_gradients(zip(gradients, variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`
- Normally, avoid grads *calculation* inside tape's *context*
- Can also have 2 different tapes for different optimization routines (as in GAN)

# `tf.GradientTape` resources

CarnegieMellon

- Play more with code:
  - tf.GradientTape API
  - Tutorial: Auto Differentiation
  - `GradientTape` with MNIST

# Outline

CarnegieMellon

1. Input
2. Auto Differentiation
3. Choosing Losses
4. Choosing Metrics

# How to apply a loss



tf_keras_loss

```
1   model.compile(optimizer=...,
2                 loss=tf.keras.losses.SparseCategoricalCrossentropy(),
3                 metrics=...)
```

- Apply a built-in loss from `tf.keras.losses`

# How to apply a loss



tf_keras_loss

```
1  model.compile(optimizer=...,
2                loss=tf.keras.losses.SparseCategoricalCrossentropy(),
3                metrics=...)
```

- Apply a built-in loss from `tf.keras.losses`
- Convenient way: apply into `model.compile` function

# How to apply a loss

Carnegie Mellon

tf_keras_loss_2

```
1  loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
2  train_loss = tf.keras.metrics.Mean(name='train_loss')
3
4  def train_step(images, labels):
5    with tf.GradientTape() as tape:
6      predictions = model(images)
7      loss = loss_object(labels, predictions) # calculate loss
8    gradients = tape.gradient(loss, model.trainable_variables)
9    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
10   train_loss(loss) # averaging
11
12 for epoch in range(EPOCHS):
13     train_loss.reset_states()
14     for images, labels in train_ds:
15         train_step(images, labels)
```

- Apply a built-in loss from `tf.keras.losses`
- Convenient way: apply into `model.compile` function
- Or in a custom (and verbose) way: by a class or function call
- Get output by `result()` function

# Common Regression Losses

Carnegie Mellon

- `MeanSquaredError`: $\|\hat{y}_i - y_i\|_2^2$
- `MeanAbsoluteError`: $|\hat{y}_i - y_i|$
- `MeanSquaredLogarithmicError`: $\|\log \hat{y}_i - \log y_i\|_2^2$
- `MeanAbsolutePercentageError`: $100 * \frac{|\hat{y}_i - y_i|}{y_i}$

# Common Classification Losses

Carnegie Mellon

- `BinaryCrossentropy`: $-y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$
- `CategoricalCrossentropy`:

$$-\sum_{i=1}^{C} y_i \log \hat{y}_i$$

  With the predicted probability for class $i \in \{1, \ldots, C\}$ calculated by $softmax$ (activation) function for output $o_i$:

$$\hat{y}_i = \frac{\exp(o_i)}{\sum_{j=1}^{C} \exp(o_j)}$$

- `SparseCategoricalCrossentropy`: probably the most used one

# Other Losses[2]

Carnegie Mellon

- `CosineSimilarity`: $-\frac{\|\hat{y}\|_2 * \|y\|_2}{\text{n_rows}}$ where L2 norm is row-wise.
- `KLDivergence`: $y * \log \frac{y}{\hat{y}}$
- `Hinge`: $\max(1 - y * \hat{y}, 0)$
- `Huber`: MSE if $|y - \hat{y}| \leq \delta$ or $\delta(\text{MAE} - \delta/2)$ otherwise
- `LogCosh, Possion, ...`

[2] https://www.tensorflow.org/api_docs/python/tf/keras/losses

# Outline

1. Input
2. Auto Differentiation
3. Choosing Losses
4. Choosing Metrics

# How to apply a metric



tf_keras_metric

```
1 model.compile(optimizer=...,
2               loss=tf.keras.losses.SparseCategoricalCrossentropy(),
3               metrics=tf.keras.metrics.Accuracy())
```

- Apply a built-in loss from `tf.keras.metrics`

# How to apply a metric

CarnegieMellon

tf_keras_metric

```
1  model.compile(optimizer=...,
2                loss=tf.keras.losses.SparseCategoricalCrossentropy(),
3                metrics=tf.keras.metrics.Accuracy())
```

- Apply a built-in loss from `tf.keras.metrics`
- Convenient way: apply into `model.compile` function

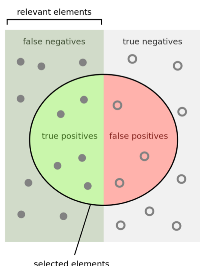
# How to apply a metric

Carnegie Mellon

```
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

@tf.function
def train_step(images, labels):
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = loss_object(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_accuracy(labels, predictions)

for epoch in range(EPOCHS):
  train_loss.reset_states()
  train_accuracy.reset_states()

  for images, labels in train_ds:
    train_step(images, labels)
```

- Other ways: like losses, but for `tf.keras.metrics`
- Also contain the most common losses

# Common metrics for binary case[3]




How many selected items are relevant?

Precision =

How many relevant items are selected?

Recall =

- TrueNegatives, TruePositives, FalseNegatives, FalsePositives
- Precision: $\frac{\text{TP}}{\text{TP+FP}}$
- Recall: $\frac{\text{TP}}{\text{TP+FN}}$
- BinaryAccuracy: $\frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$

[3] Those metrics can be generalized for multi-classes as well.

Image credit: wikipedia

# Binary cases: ROC Curve[4]

Carnegie Mellon

| A | | | B | | | C | | |
|---|---|---|---|---|---|---|---|---|
| TP=63 | FP=28 | 91 | TP=77 | FP=77 | 154 | TP=24 | FP=88 | 112 |
| FN=37 | TN=72 | 109 | FN=23 | TN=23 | 46 | FN=76 | TN=12 | 88 |
| 100 | 100 | 200 | 100 | 100 | 200 | 100 | 100 | 200 |
| TPR = 0.63 | | | TPR = 0.77 | | | TPR = 0.24 | | |
| FPR = 0.28 | | | FPR = 0.77 | | | FPR = 0.88 | | |
| PPV = 0.69 | | | PPV = 0.50 | | | PPV = 0.21 | | |
| F1 = 0.66 | | | F1 = 0.61 | | | F1 = 0.23 | | |
| ACC = 0.68 | | | ACC = 0.50 | | | ACC = 0.18 | | |

- Consider A, B, and C, which one is the best?

[4] https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Image credit: wikipedia

# Binary cases: ROC Curve



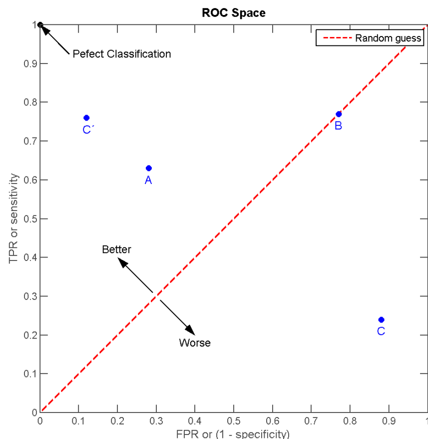
| A | | | B | | | C | | | C′ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TP=63 | FP=28 | 91 | TP=77 | FP=77 | 154 | TP=24 | FP=88 | 112 | TP=76 | FP=12 | 88 |
| FN=37 | TN=72 | 109 | FN=23 | TN=23 | 46 | FN=76 | TN=12 | 88 | FN=24 | TN=88 | 112 |
| 100 | 100 | 200 | 100 | 100 | 200 | 100 | 100 | 200 | 100 | 100 | 200 |
| TPR = 0.63 | | | TPR = 0.77 | | | TPR = 0.24 | | | TPR = 0.76 | | |
| FPR = 0.28 | | | FPR = 0.77 | | | FPR = 0.88 | | | FPR = 0.12 | | |
| PPV = 0.69 | | | PPV = 0.50 | | | PPV = 0.21 | | | PPV = 0.86 | | |
| F1 = 0.66 | | | F1 = 0.61 | | | F1 = 0.23 | | | F1 = 0.81 | | |
| ACC = 0.68 | | | ACC = 0.50 | | | ACC = 0.18 | | | ACC = 0.82 | | |

- Consider A, B, and C, which one is the best?
- How about C', the inverse of C?

Image credit: wikipedia

# Binary cases: ROC Curve



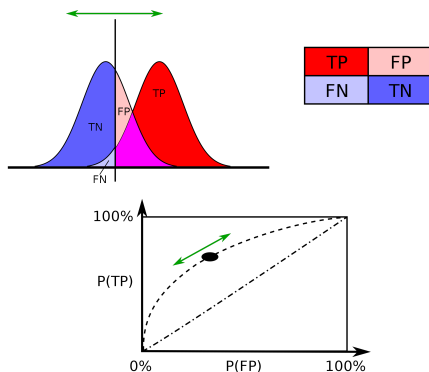



- How about C', the inverse of C?

Image credit: wikipedia

# Binary cases: ROC Curve

Carnegie Mellon

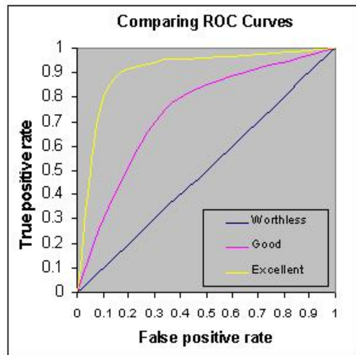


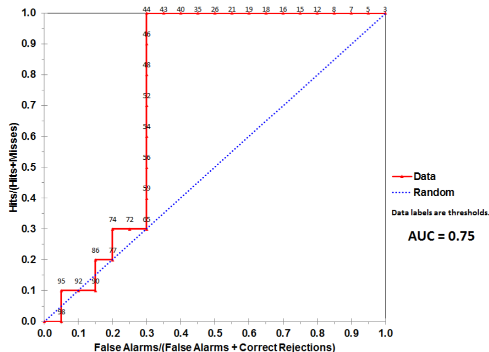

- Tradeoff between $\text{TPR} = \frac{\text{TP}}{\text{TP+FN}}$ (`Recall`) and $\text{FPR} = \frac{\text{FP}}{\text{FP+TN}}$
- These rates are *stable* metrics with classification errors

Image credit: `wikipedia`

# Binary cases: AUC

CarnegieMellon




- AUC: Area under (the ROC) curve, ranges $[0, 1]$
  - AUC is the "aggregation" of ROC over all thresholds
  - A single, stable metric combining others, and so is desirable

Image credit: gim.unmc.edu and wikipedia

# Common metrics for general cases[5]



- `Accuracy`: across all classes, can be optionally weighted, for scalars
- `CategoricalAccuracy`: general cases, for one-hots labels
- `SparseCategoricalAccuracy`: for scalar labels
- `TopKCategoricalAccuracy`: for $k > 1$
- `SparseTopKCategoricalAccuracy`: for scalar labels

[5] More: `https://www.tensorflow.org/api_docs/python/tf/keras/metrics`