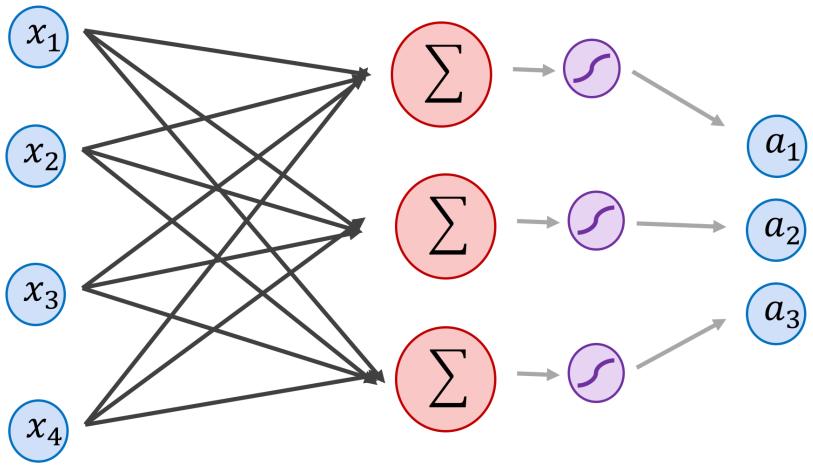
11-695: AI Engineering
Feed-forward NNs

LTI/SCS

Spring 2020

Outline

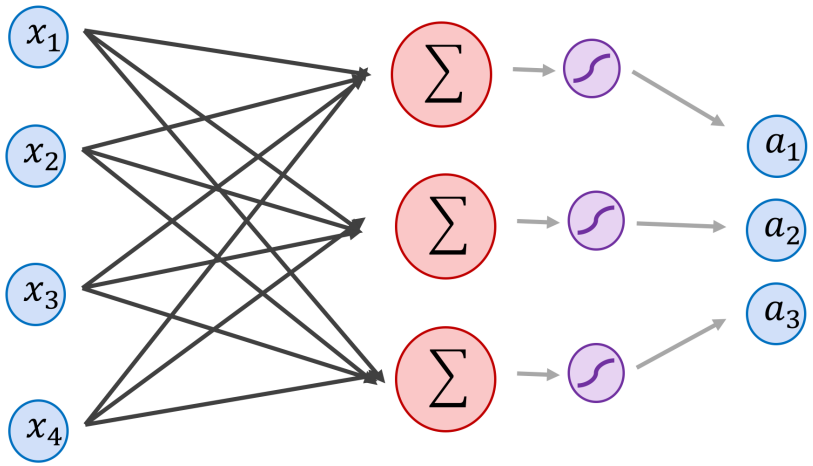
Construction of NNs



- Two basic operations
  - Linear: $o_i = W_i^T a_{i-1} + b_i$
  - Nonlinear (by activation functions) : $a_i = \phi(o_i)$

Image credit: Vicente Ordóñez Román

Construction of NNs



- Two basic operations
  - Linear: $o_i = W_i^T a_{i-1} + b_i$
  - Nonlinear (by activation functions) : $a_i = \phi(o_i)$
- Usually comprise of a sequence of such pair of basic operations
- Mathematically:

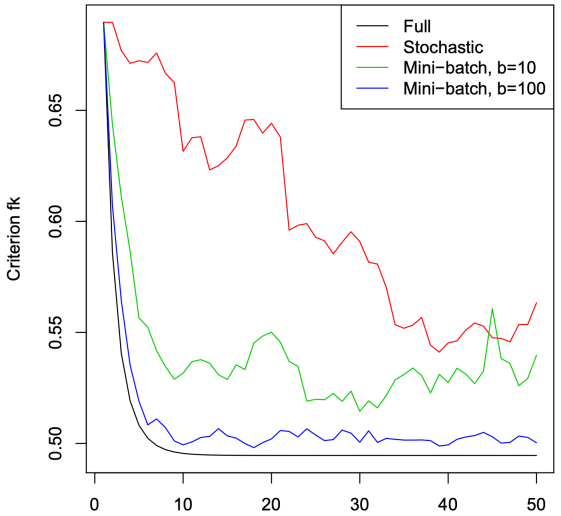
$$\hat{\mathbf{y}} = (\phi_n \circ \mathbf{f}_{W_n} \circ \phi_{n-1} \circ \mathbf{f}_{W_{n-1}} \ldots \phi_1 \circ \mathbf{f}_{W_1})(\mathbf{X})$$

$$\hat{\mathbf{W}} = \{W_n, W_{n-1}, \ldots, W_1\} = \underset{\mathbf{W}}{\operatorname{argmin}}\, \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$$

Image credit: Vicente Ordóñez Román

Stochastic Gradient Descent (SGD)

Carnegie Mellon

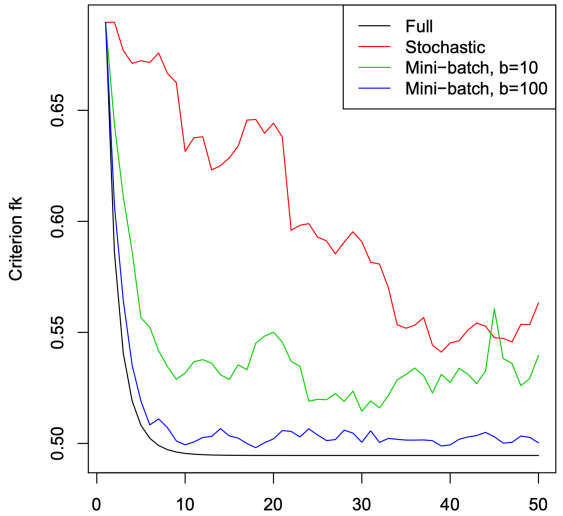


- Use gradient of params to update params themselves with a step

Image credit: Ryan Tibshirani

Stochastic Gradient Descent (SGD)

Carnegie Mellon



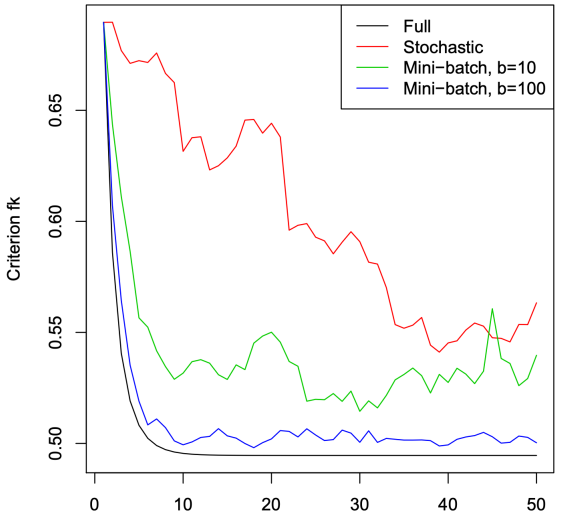
- Use gradient of params to update params themselves with a step
- How to calculate gradients for this NN?

$$\hat{\mathbf{y}} = (\phi_n \circ \mathbf{f}_{W_n} \circ \phi_{n-1} \circ \mathbf{f}_{W_{n-1}} \ldots \phi_1 \circ \mathbf{f}_{W_1})(\mathbf{X})$$

$$\hat{\mathbf{W}} = \{W_n, W_{n-1}, \ldots, W_1\} = \underset{\mathbf{W}}{\operatorname{argmin}}\, \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$$

Image credit: Ryan Tibshirani

Stochastic Gradient Descent (SGD)



- Use gradient of params to update params themselves with a step
- How to calculate gradients for this NN?

$$\hat{\mathbf{y}} = (\phi_n \circ \mathbf{f}_{W_n} \circ \phi_{n-1} \circ \mathbf{f}_{W_{n-1}} \ldots \phi_1 \circ \mathbf{f}_{W_1})(\mathbf{X})$$

$$\hat{\mathbf{W}} = \{W_n, W_{n-1}, \ldots, W_1\} = \underset{\mathbf{W}}{\operatorname{argmin}}\, \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$$

- Answer: *chain rule*

Image credit: Ryan Tibshirani

Chain Rule Calculus

Carnegie Mellon

- Assume

$$z = (\mathbf{f} \circ \mathbf{g})(x)$$

- In detail with 2 sequential steps

$$y = \mathbf{g}(x),$$
$$z = \mathbf{f}(y)$$

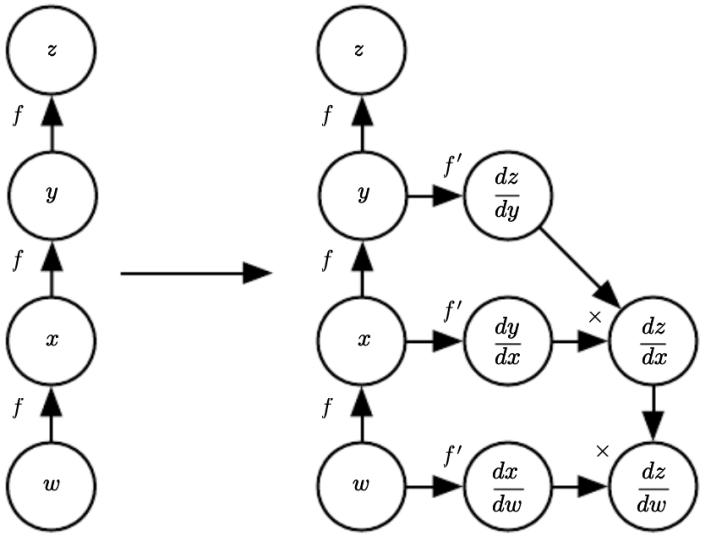
- Then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Image credit: Ryan Tibshirani

Backpropagation (Rumelhart)[1]

Carnegie Mellon

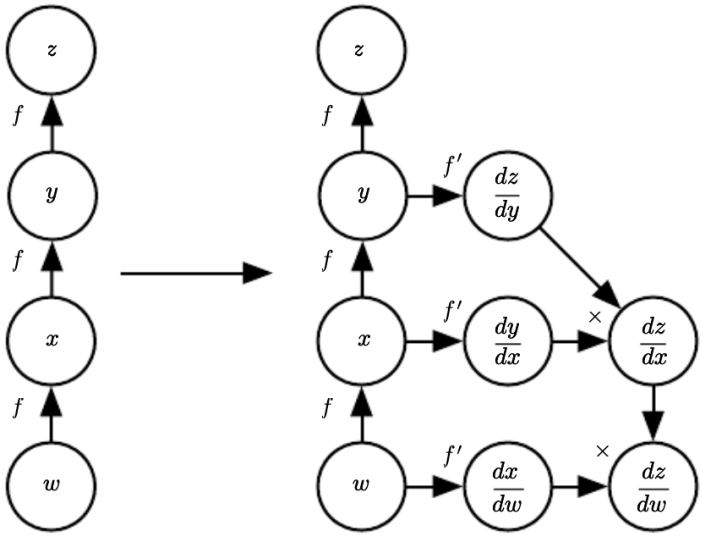


- Recursively apply the chain rule from *loss* back to *weights*

[1] https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Image credit: Ian Goodfellow *el al.*

Backpropagation (Rumelhart)[1]



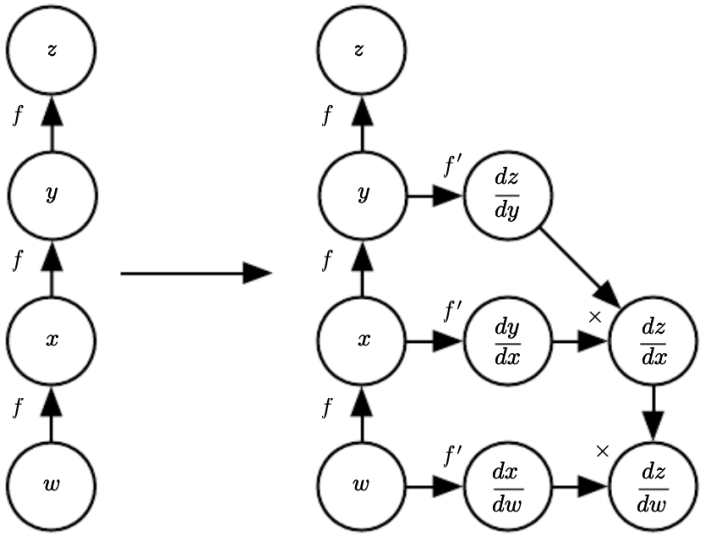
- Recursively apply the chain rule from *loss* back to *weights*
- Via 2 small steps
  - Forward: Calculate loss from a batch
  - Backward: Calculate params' gradients

[1] https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Image credit: Ian Goodfellow *el al.*

Backpropagation (Rumelhart)[1]

Carnegie Mellon



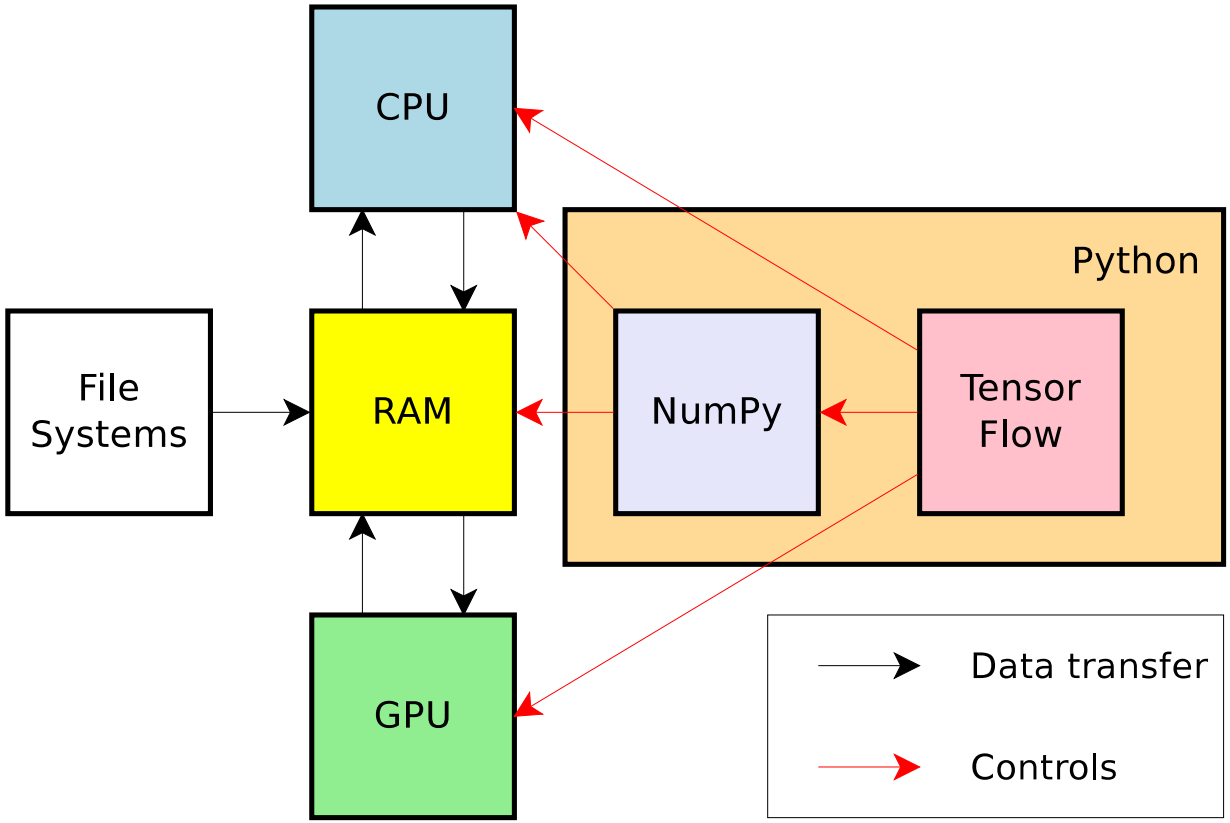
- Recursively apply the chain rule from *loss* back to *weights*
- Via 2 small steps
  - Forward: Calculate loss from a batch
  - Backward: Calculate params' gradients
- In practice, do we have to calculate gradients step by step?

[1] https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

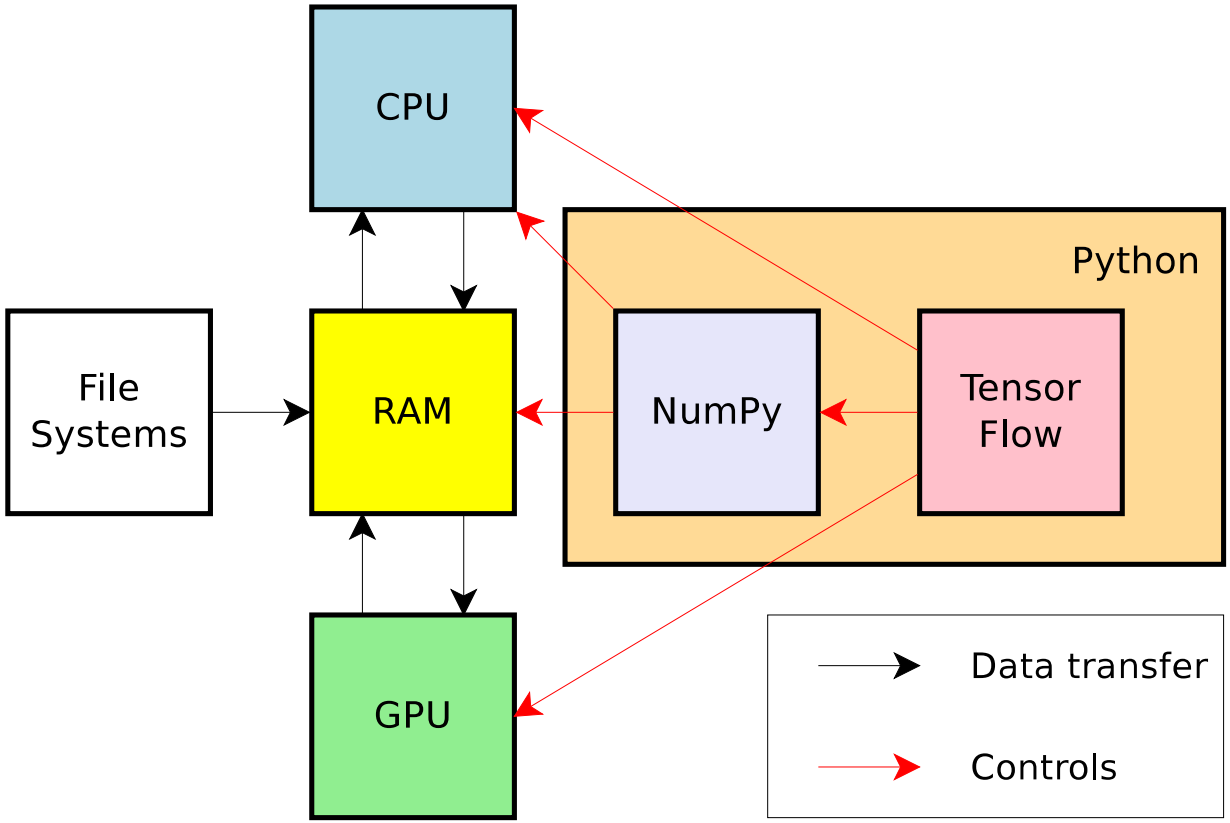
Image credit: Ian Goodfellow *el al.*

Outline

Overview

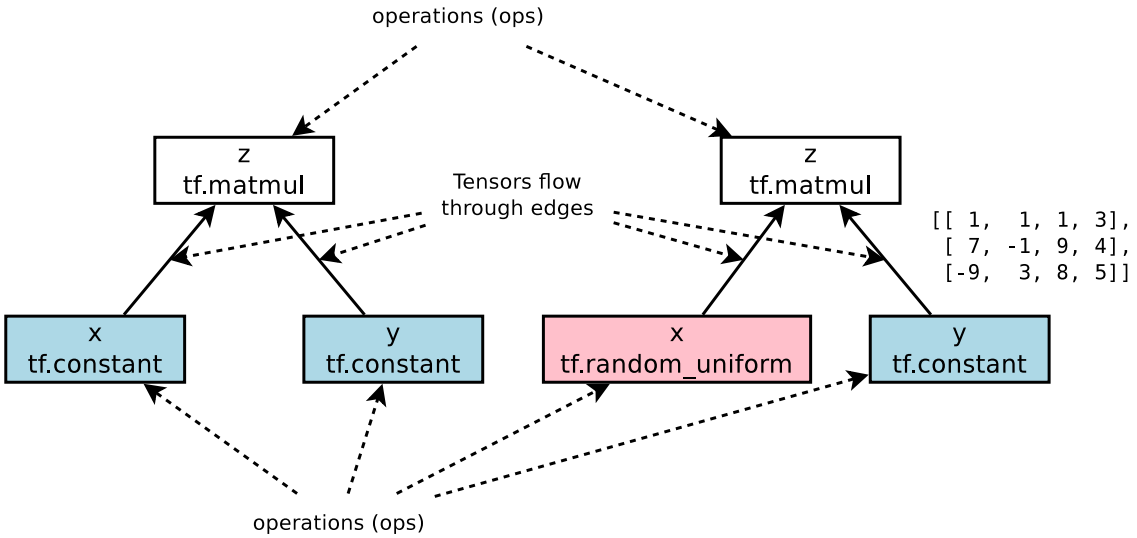

CPU
Python
File Systems
RAM
NumPy
Tensor Flow
GPU
Data transfer
Controls

Overview



- It's important to distinguish between *tensors* and other Python/Numpy data.

Computational Graph



- `tf` computational graph is a *directed acyclic graph* (DAG)
  - Nodes are called *operations*, or *ops*
  - Variables, constants are also ops
  - Ops produce *tensors*
  - Tensors flow around through edges
- It performs *auto-differentiation*

Variables

Carnegie Mellon

tf_variable

```
1  import tensoflow as tf
2
3  x = tf.Variable(initial_value=tf.ones(shape=(2,2)), name='W')
4  # <tf.Variable 'W:0' shape=(2, 2) dtype=float32, numpy=
5  # array([[1., 1.],
6  #        [1., 1.]], dtype=float32)>
7
8  x_np = x.numpy() # tensor to numpy values
9
10 y = tf.constant([1, 2, 3, 4, 5, 6], shape=(2,3))
11 # <tf.Tensor: id=110, shape=(2, 3), dtype=int32, numpy=
12 # array([[1, 2, 3],
13 #        [4, 5, 6]], dtype=int32)>
```

- Variables and constants are inputs

Variables

Carnegie Mellon

tf_variable

```
1  import tensoflow as tf
2
3  x = tf.Variable(initial_value=tf.ones(shape=(2,2)), name='W')
4  # <tf.Variable 'W:0' shape=(2, 2) dtype=float32, numpy=
5  # array([[1., 1.],
6  #        [1., 1.]], dtype=float32)>
7
8  x_np = x.numpy() # tensor to numpy values
9
10 y = tf.constant([1, 2, 3, 4, 5, 6], shape=(2,3))
11 # <tf.Tensor: id=110, shape=(2, 3), dtype=int32, numpy=
12 # array([[1, 2, 3],
13 #        [4, 5, 6]], dtype=int32)>
```

- Variables and constants are inputs
- In graph, they have no parents, but other ops

Outline

Build a Feed-forward NN

Carnegie Mellon

tf_model

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  # TensorFlow and tf.keras
4  import tensorflow as tf
5  from tensorflow import keras
6
7  model = keras.Sequential([
8      keras.layers.Flatten(input_shape=(28, 28)),
9      keras.layers.Dense(128, activation='relu'),
10     keras.layers.Dense(64, activation='relu'),
11     keras.layers.Dense(10, activation='softmax')
12 ])
```

- Use `tf.keras.Seqential`

Build a Feed-forward NN

Carnegie Mellon

tf_model

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  # TensorFlow and tf.keras
4  import tensorflow as tf
5  from tensorflow import keras
6
7  model = keras.Sequential([
8      keras.layers.Flatten(input_shape=(28, 28)),
9      keras.layers.Dense(128, activation='relu'),
10     keras.layers.Dense(64, activation='relu'),
11     keras.layers.Dense(10, activation='softmax')
12 ])
```

- Use `tf.keras.Seqential`
- Each hidden layer use `Dense` with the *number of output neurons*

Build a Feed-forward NN

Carnegie Mellon

tf_model

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  # TensorFlow and tf.keras
4  import tensorflow as tf
5  from tensorflow import keras
6
7  model = keras.Sequential([
8      keras.layers.Flatten(input_shape=(28, 28)),
9      keras.layers.Dense(128, activation='relu'),
10     keras.layers.Dense(64, activation='relu'),
11     keras.layers.Dense(10, activation='softmax')
12 ])
```

- Use `tf.keras.Seqential`
- Each hidden layer use `Dense` with the *number of output neurons*
- For feed-forward, must `Flatten` or `Reshape` to a vector

Build a Feed-forward NN

Carnegie Mellon

tf_model_2

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  import tensorflow as tf
4  from tensorflow import keras
5
6  class MyModel(keras.Model):
7    def __init__(self):
8      super(MyModel, self).__init__()
9      self.flatten = keras.layers.Flatten()
10     self.d1 = keras.layers.Dense(128, activation='relu')
11     self.d2 = keras.layers.Dense(64, activation='relu')
12     self.d3 = keras.layers.Dense(10, activation='softmax')
13   def call(self, x):
14     x = self.flatten(x)
15     x = self.d1(x)
16     x = self.d2(x)
17     return self.d3(x)
18
19 # Create an instance of the model
20 model = MyModel()
```

- Can also extend from `tf.keras.Model` class

Outline

Make model ready for training

Carnegie Mellon

keras_compile

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  import tensorflow as tf
4  from tensorflow import keras
5
6  model.compile(optimizer='adam',
7                loss='sparse_categorical_crossentropy',
8                metrics=['accuracy'])
```

- Use `model.compile()`

Make model ready for training

Carnegie Mellon

keras_compile

```
1  from __future__ import absolute_import, division, print_function, unicode_literals
2
3  import tensorflow as tf
4  from tensorflow import keras
5
6  model.compile(optimizer='adam',
7                loss='sparse_categorical_crossentropy',
8                metrics=['accuracy'])
```

- Use `model.compile()`
- Must have *optimizer*, *loss* and *metrics* types beforehand

Training, Testing and Predicting

Carnegie Mellon

keras_train

```
1  model.fit(train_images, train_labels, epochs=10)
2
3  # testing (evaluating)
4  test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
5
6  # predicting
7  predictions = model.predict(test_images)
```

- Use `model.fit()` for training
- Use `model.evaluate()` for testing
- Use `model.predict()` for testing

Training, Testing and Predicting

Carnegie Mellon

keras_train

```
1  model.fit(train_images, train_labels, epochs=10)
2
3  # testing (evaluating)
4  test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
5
6  # predicting
7  predictions = model.predict(test_images)
```

- Use `model.fit()` for training
- Use `model.evaluate()` for testing
- Use `model.predict()` for testing
- All other complication is covered by `keras`, but it's open for customization.

Training, Testing and Predicting

Carnegie Mellon

keras_train

```
1  model.fit(train_images, train_labels, epochs=10)
2
3  # testing (evaluating)
4  test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
5
6  # predicting
7  predictions = model.predict(test_images)
```

- Use `model.fit()` for training
- Use `model.evaluate()` for testing
- Use `model.predict()` for testing
- All other complication is covered by `keras`, but it's open for customization.
- Run yourself: ▸ Demo