STAT679
Computing for Data Science and Statistics

Lecture 25: TensorFlow, continued

TensorFlow

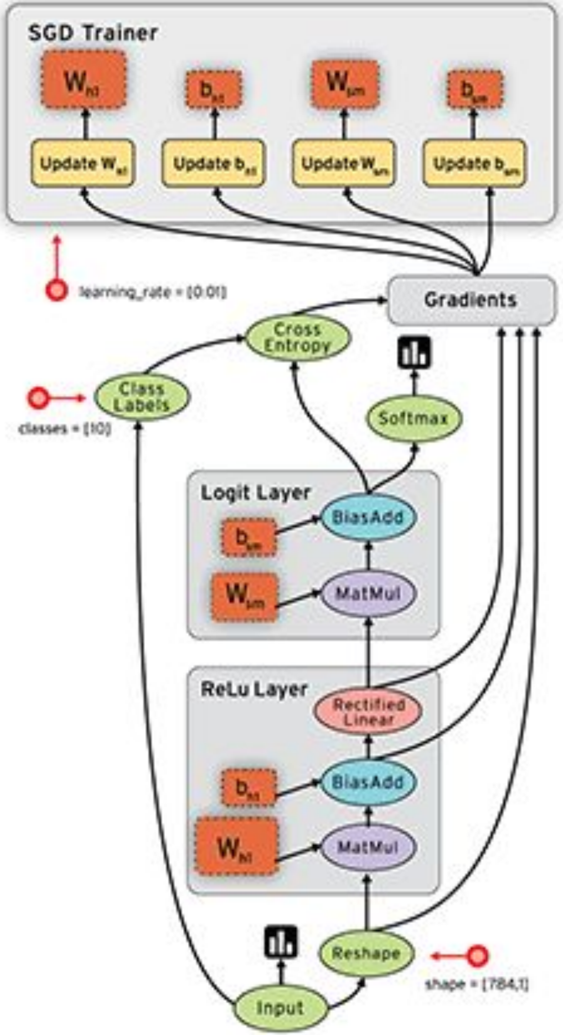
Previous lecture: Introduction to TensorFlow

- `tf.Tensor` objects represent tensors
- Tensors are combined into a computational graph
    - Captures the computational operations to be carried out at runtime

This lecture: Advanced TF

- More detail on the computational graph and `tf.Tensor` objects
- **Lab:** recognizing MNIST handwritten digits

Recall: TensorFlow as DataFlow



Computational graph: how data “flows” through program

In previous lecture:

We were a bit fast and loose with nodes and edges

Strictly speaking:

Nodes are operations (`tf.Operation`)

Edges are tensors (`tf.Tensor`)

More on the Computational Graph

`tf.Graph`

- Special class provided by TF to represent a computational graph
- Contains `tf.Operation` objects and `tf.Tensor` objects
  - ...and keeps track of how they interact (i.e., the graph structure itself)

As of TF version 2, working with `tf.Graph` directly is deprecated

- Instead we use `tf.function` objects
- But there is still a `tf.Graph` object lurking behind the scenes!

More: https://www.tensorflow.org/api_docs/python/tf/Graph

- https://www.tensorflow.org/api_docs/python/tf/function

More on the Computational Graph

`tf.Tensor`

(Already familiar to you)

Represents a tensor, i.e., data on which to perform computations

`tf.Operation`

TF class that represents a computation performed on zero or more tensors

Encoded as a node in a computational graph

Tensor operations

Previous lecture: we saw different ways of creating tensors…

...but not much in the way of how to do things with them.

Example functions available in TF:

Math operations (trigonometric functions, special functions, logicals)

Matrix operations (matrix-vector multiplication, decompositions)

Reduce operations (e.g., summing or taking the mean along an axis)

Tensor operations: +,-,*,/

```python
import tensorflow as tf

a = tf.constant(5, dtype=tf.float32)
b = tf.constant(3.1415, dtype=tf.float32)
c = tf.constant(2, dtype=tf.float32)

def silly_pyfunction(x,y,z):
    return x/a + b*y - c*z
silly = tf.function(silly_pyfunction)

print(silly([4,3,2,1], [2,3,4,5], [1,1,2,2]))
```

```
tf.Tensor([ 5.083   8.0245  8.966  11.9075], shape=(4,), dtype=float32)
```

`+,-,*,/` short for `tf.add()`, `tf.subtract()`, `tf.multiply()`, `tf.divide()`, respectively.

```python
x = tf.constant(1, dtype=tf.float32)
y = tf.constant(0, dtype=tf.float32)
x/y
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=inf>
```

Note: Division by zero results in `inf`, rather than `nan`.

Matrix multiplication in TF: `tf.matmul()`

```
M = tf.constant([[1,0,1],[0,1,1],[1,1,0]], dtype=tf.float32)
oneThruNine = tf.constant([[1,2,3],[4,5,6],[7,8,9]], dtype=tf.float32)
c = tf.matmul(oneThruNine, M)
print( c.numpy() )
```

```
[[ 4.  5.  3.]
 [10. 11.  9.]
 [16. 17. 15.]]
```

`tf.matmul(A,B)` multiplies tensors `A` and `B`, as matrices, provided their ranks and types agree.

```
M1 = tf.constant([[1,0,1],[0,1,1]], dtype=tf.float32)
M2 = tf.constant([[1,0,1,1],[0,0,1,1]], dtype=tf.float32)
R = tf.matmul(M1,M2)
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-9-715b0435c363> in <module>
      1 M1 = tf.constant([[1,0,1],[0,1,1]], dtype=tf.float32)
      2 M2 = tf.constant([[1,0,1,1],[0,0,1,1]], dtype=tf.float32)
----> 3 R = tf.matmul(M1,M2)
        ...

~/.local/lib/python3.8/site-packages/six.py in raise_from(value, from_value)

InvalidArgumentError: Matrix size-incompatible: In[0]: [2,3], In[1]: [2,4] [Op:MatMul]
```

Matrix multiplication in TF: `tf.matmul()`

```python
M = tf.constant([[1,0,1],[0,1,1],[1,1,0]], dtype=tf.float32)
oneThruNine = tf.constant([[1,2,3],[4,5,6],[7,8,9]], dtype=tf.float32)
c = tf.matmul(oneThruNine, M)
print( c.numpy() )
```

```
[[ 4.  5.  3.]
 [10. 11.  9.]
 [16. 17. 15.]]
```

`tf.matmul(A,B)` multiplies tensors `A` and `B`, as matrices, provided their ranks and types agree.

```python
M1 = tf.constant([[1,0,1],[0,1,1]], dtype=tf.float32)
M2 = tf.constant([[1,0,1,1],[0,0,1,1]], dtype=tf.float32)
R = tf.matmul(M1,M2)
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-9-715b0435c363> in <module>
      1 M1 = tf.constant([[1,0,1],[0,1,1]], dtype=tf.float32)
      2 M2 = tf.constant([[1,0,
----> 3 R = tf.matmul(M1,M2)
        ...

~/.local/lib/python3.8/site-pac

InvalidArgumentError: Matrix size-incompatible: In[0]: [2,3], In[1]: [2,4] [Op:MatMul]
```

Note: `tf.matmul()` can be used to multiply tensors of arbitrary rank. Using appropriate flags, we can transpose/adjoint the arguments as we please.
https://www.tensorflow.org/api_docs/python/tf/linalg/matmul

More matrix operations in TF: `tf.linalg`

`tf.linalg.diag`: picks out diagonal of a matrix (or other tensor)

`tf.linalg.det`: computes determinant of a matrix

`tf.linalg.inv`: computes inverse of a matrix

`tf.linalg.solve`: solves Ax = b

`tf.linalg.matrix_transpose`: transposes a matrix

`tf.linalg.cholesky(...)`: computes Cholesky decomposition
https://en.wikipedia.org/wiki/Cholesky_decomposition

Element-wise operations in TF

TF element-wise operations are just like Numpy universal functions

Examples:

`tf.math.abs()`: computes absolute value

`tf.math.acos()`: computes arccosine

`tf.math.cos()`: computes cosine

`tf.math.exp()`: computes exponential

`tf.math.log()`: computes logarithm

`tf.math.sigmoid()`: computes sigmoid function

https://en.wikipedia.org/wiki/Sigmoid_function

```
r = tf.constant(-5, dtype=tf.float32)
c = tf.constant(1+1j, dtype=tf.complex64)

print( tf.math.abs(r) )
print( tf.math.abs(c) )
```

```
tf.Tensor(5.0, shape=(), dtype=float32)
tf.Tensor(1.4142135, shape=(), dtype=float32)
```

Element-wise comparisons in TF

TF supports element-wise comparisons of tensors

- `tf.math.less()`, `tf.math.less_equal()`,
- `tf.math.greater()`, `tf.math.greater_equal()`
- `tf.math.equal()`, `tf.math.not_equal()`

Logical (operate on tensors with `dtype=bool`)

- `tf.math.logical_and()`
- `tf.math.logical_or()`
- `tf.math.logical_xor()`
- Also supported: `tf.math.logical_not()`, but this isn't a comparison

So, TF has a lot of stuff going on!

“low-level” TF API makes lots of powerful tools available

...almost too many!

I just wanted to train a neural net!
Why do I have to worry about all this stuff?!

Rest of Lecture: Lab

1) We'll use softmax regression to classify handwritten digits
    Using the low-level API that we discussed last lecture

2) We'll build and train a convolutional NN on the same data
    Using the `tf.keras` API, which hides much of the low-level operations

Workshop: Recognizing MNIST Digits

MNIST is a famous computer vision data set

28-by-28 greyscale images of hand-written digits

https://en.wikipedia.org/wiki/MNIST_database

Each image is labeled according to what digit it represents

2012: 0.23 percent error rate: https://arxiv.org/abs/1202.2745

(there has probably been improvement in this number since then…)

Pared-down demo code:

http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2022/STAT679/democode/softmax_mnist_demo.ipynb

Image credit: Wyss, König, and Verschure (2003)

Recognizing MNIST Digits

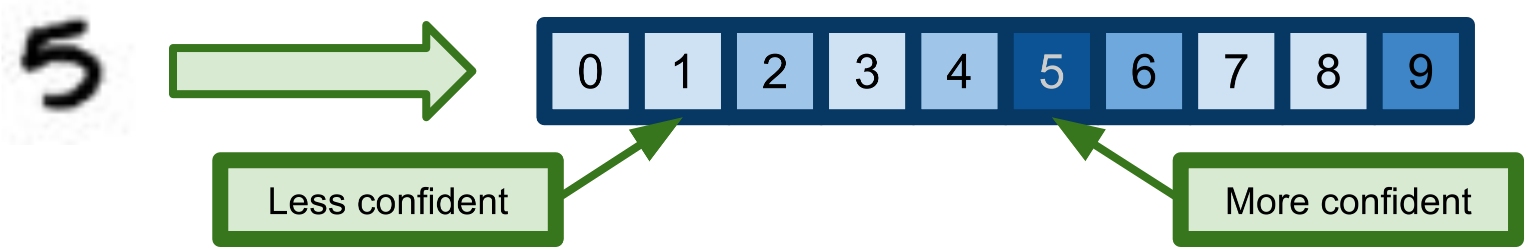
Goal: given an image, classify what digit it represents.

= 5?

= 9?

In particular, we'll build a model that outputs a vector of probabilities

`i`-th entry of vector will be model's confidence that image is digit `i`.



Softmax Regression

Generalizes logistic regression to categorical variables with >2 values

Softmax function: $\sigma_j(z) = \frac{e^{z_j}}{\sum_i e^{z_i}}$

Probability that the observation is from category **j**

Model parameters

Our model will assign probabilities to digits as $\mathbb{P}[Y = j] = \sigma_j(WX + b)$

Observed data

More information:
https://en.wikipedia.org/wiki/Multinomial_logistic_regression
https://en.wikipedia.org/wiki/Softmax_function
C. M. Bishop (2006). *Pattern Recognition and Machine Learning*. Springer.

The Plan

Represent 28-by-28 images by flattened 784-dimensional vectors

Apply softmax regression to vectors

  Learn weights `W` and bias `b`

  Train on a training set of labeled images

Evaluate learned model on test set

Flattening the data

Images are most naturally represented as matrices...

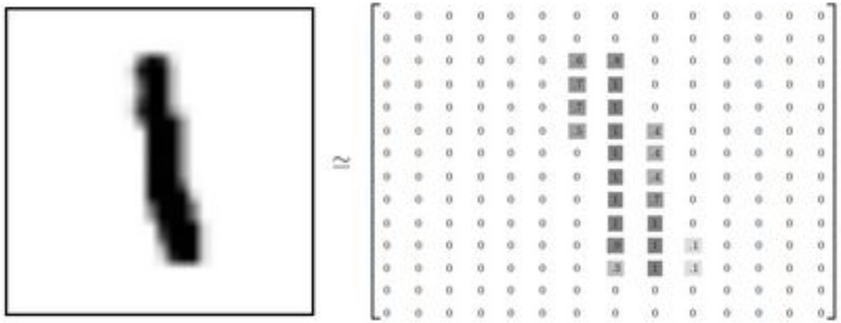


Image credit: TensorFlow tutorial

...but softmax regression requires vector inputs.

Solution: “unroll” image into a vector. It doesn’t matter how we do this, so long as we’re consistent. That is, so long as every image is flattened to a vector in the same **same way**.

Building the model

```python
class SoftmaxModel(tf.Module):
    def __init__(self, d_data, d_class):
        super().__init__()
        self.W = tf.Variable(tf.random.normal(shape=[d_class,d_data]), dtype=tf.float32)
        self.b = tf.Variable(tf.random.normal(shape=[d_class]), dtype=tf.float32)
    def __call__(self, x):
        z = tf.linalg.matvec( self.W, x) + self.b
        return tf.nn.softmax( z )

classifier = SoftmaxModel( 784, 10 )
```

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Image credit: TensorFlow v1 tutorial

Building the model

Our model outputs a 10-dimensional probability. So `W` should map a vector to a 10-vector.

```python
class SoftmaxModel(tf.Module):
    def __init__(self, d_data, d_class):
        super().__init__()
        self.W = tf.Variable(tf.random.normal(shape=[d_class,d_data]), dtype=tf.float32)
        self.b = tf.Variable(tf.random.normal(shape=[d_class]), dtype=tf.float32)
    def __call__(self, x):
        z = tf.linalg.matvec( self.W, x) + self.b
        return tf.nn.softmax( z )

classifier = SoftmaxModel( 784, 10 )
```

Bias term is same dimension as `Wx`.

Each row of `x` is going to be a single observation, a 784-dimensional vector (28-by-28 image has 784 pixels).

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Image credit: TensorFlow v1 tutorial

Training the model: choosing a loss function

To train our model, we need to choose a loss function

We’ll use cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy

Related to the KL divergence

$$H_{y'}(y) = \underbrace{\sum_i}_{\text{Sum over digits 0 to 9}} \underbrace{y'_i}_{\text{The true distribution}} \log \underbrace{y_i}_{\text{Our model}}$$

Training the model: choosing a loss function

To train our model, we need to choose a loss function

We'll use cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy

Related to the KL divergence

$$H_{y'}(y) = \sum_i y'_i \log y_i$$

Sum over digits 0 to 9

The true distribution

Our model

Note: the formula above is the sum for **one** observation. Our actual loss function will be a sum of these sums: for each training example, we need to sum of over the 10 digits.

Training the model: building more of the graph

We'll read the truth into `ytrue`, while `y_pred` will be our model's predicted labels.

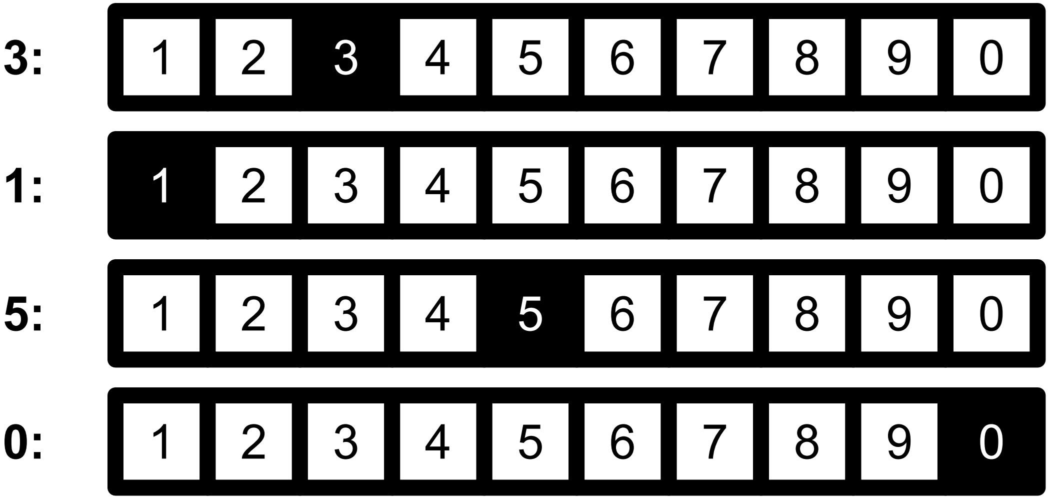
```
def crossent_loss(y_true, y_pred):
    crossents = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
    return tf.reduce_mean( crossents )
```

Note: we are using what is called a **one-hot** encoding in the true labels `ytrue`.

Aside: one-hot encodings

In ML, it is common to represent categorical variables by vectors

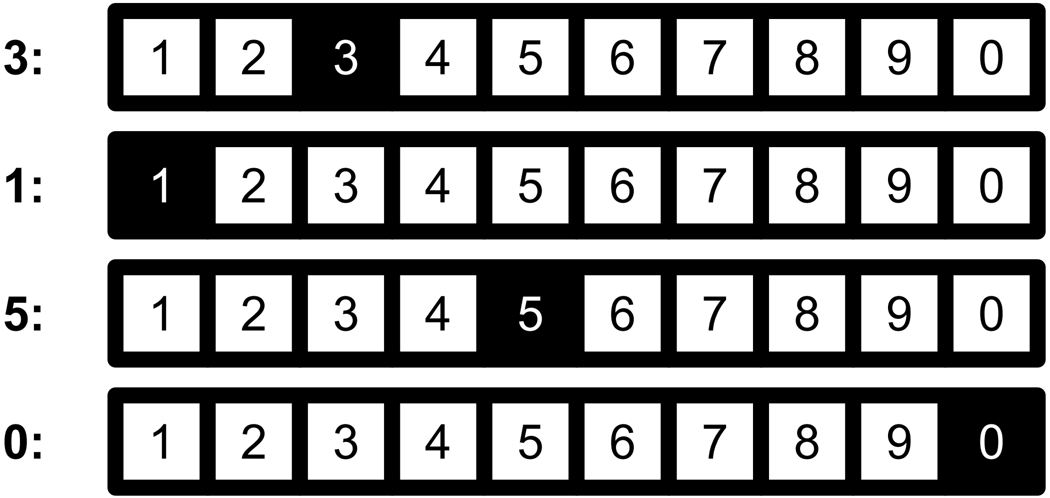
- K possible values for the variable
- represent by a K-dimensional vector
- Object of k-th category represented by vector with k-th entry 1, rest 0



Aside: one-hot encodings

In ML, it is common to represent categorical variables by vectors

- K possible values for the variable
- represent by a K-dimensional vector
- Object of k-th category represented by vector with k-th entry 1, rest 0



Note: this is a case where it's good to use the `tf.SparseTensor` object. If K is really big, it's expensive to store all those 0s! In our application, K=10, so it's no big deal, but in, for example, NLP, K=1e6 is not uncommon.

Training the model

To train our model, we need to choose a loss function

We'll use cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy

Related to the KL divergence

$$H_{y'}(y) = \sum_i y'_i \log y_i$$

```
def crossent_loss(y_true, y_pred):
    crossents = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
    return tf.reduce_mean( crossents )
```

Training the model

Take the gradient of the loss with respect to the model parameters, update accordingly. This pattern should look familiar from the previous lecture.

```
def train(model, images, labels, learning_rate):

    with tf.GradientTape() as t:
        current_loss = crossent_loss(labels, model(images))

    dW, db = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub(learning_rate * dW)
    model.b.assign_sub(learning_rate * db)
```

```
def training_loop(model, x, y, epochs=1):
    for e in range(epochs):
        ds = tf.data.Dataset.from_tensor_slices( (x, y) )
        ds = ds.shuffle( x.shape[0] ).batch(100)
        for (xbatch,ybatch) in ds:
            train(model, xbatch, ybatch, learning_rate=0.5)
    predicted = model(x)
    train_loss = crossent_loss(y, predicted)
    train_acc = accuracy( y, predicted )
    print("Final state: train_loss=%2.5f, train_acc=%2.5f" %
              (train_loss, train_acc) )
```

Training the model

```
def train(model, images, labels, learning_rate):

    with tf.GradientTape() as t:
        current_loss = crossent_loss(labels, model(images))

    dW, db = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub(learning_rate * dW)
    model.b.assign_sub(learning_rate * db)
```

Take the gradient of the loss with respect to the model parameters, update accordingly. This pattern should look familiar from the previous lecture.

```
def training_loop(model, x, y, epochs=1):
    for e in range(epochs):
        ds = tf.data.Dataset.from_tensor_slices( (x, y) )
        ds = ds.shuffle( x.shape[0] ).batch(100)
        for (xbatch, ybatch) in ds:
            train(model, xbatch, ybatch, learning_rate=0.5)
    predicted = model(x)
    train_loss = crossent_loss(y, predicted)
    train_acc = accuracy( y, predicted )
    print("Final state: train_loss=%2.5f, train_acc=%2.5f" %
                (train_loss, train_acc) )
```

The `tf.data.Dataset` object provides tools for working with data, including shuffling and batching.

Instead of evaluating the loss on all 60K training elements for every gradient step, we are using a small subset of the data (called a **batch**).

More information: https://www.tensorflow.org/api_docs/python/tf/data/Dataset

Training the model

```python
def train(model, images, labels, learning_rate):

    with tf.GradientTape() as t:
        current_loss = crossent_loss(labels, model(images))

    dW, db = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub(learning_rate * dW)
    model.b.assign_sub(learning_rate * db)
```

Take the gradient of the loss with respect to the model parameters, update accordingly. This pattern should look familiar from the previous lecture.

Iterate over all of the batches. Each is a set of 100 (image,label) pairs.

```python
def training_loop(model, x, y, epochs=1):
    for e in range(epochs):
        ds = tf.data.Dataset.from_tensor_slices( (x, y) )
        ds = ds.shuffle( x.shape[0] ).batch(100)
        for (xbatch,ybatch) in ds:
            train(model, xbatch, ybatch, learning_rate=0.5)
    predicted = model(x)
    train_loss = crossent_loss(y, predicted)
    train_acc = accuracy( y, predicted )
    print("Final state: train_loss=%2.5f, train_acc=%2.5f" %
                 (train_loss, train_acc) )
```

Instead of evaluating the loss on all 60K training elements for every gradient step, we are using a small subset of the data (called a **batch**).

More information: https://www.tensorflow.org/api_docs/python/tf/data/Dataset

Training the model

```python
def train(model, images, labels, learning_rate):

    with tf.GradientTape() as t:
        current_loss = crossent_loss(labels, model(images))

    dW, db = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub(learning_rate * dW)
    model.b.assign_sub(learning_rate * db)
```

Take the gradient of the loss with respect to the model parameters, update accordingly. This pattern should look familiar from the previous lecture.

```python
def training_loop(model, x, y, epochs=1):
    for e in range(epochs):
        ds = tf.data.Dataset.from_tensor_slices( (x, y) )
        ds = ds.shuffle( x.shape[0] ).batch(100)
        for (xbatch,ybatch) in ds:
            train(model, xbatch, ybatch, learning_rate=0.5)
    predicted = model(x)
    train_loss = crossent_loss(y, predicted)
    train_acc = accuracy( y, predicted )
    print("Final state: train_loss=%2.5f, train_acc=%2.5f" %
                (train_loss, train_acc) )
```

Each “epoch”, we will go through the dataset once.

Instead of evaluating the loss on all 60K training elements for every gradient step, we are using a small subset of the data (called a **batch**).

More information: https://www.tensorflow.org/api_docs/python/tf/data/Dataset

Training the model

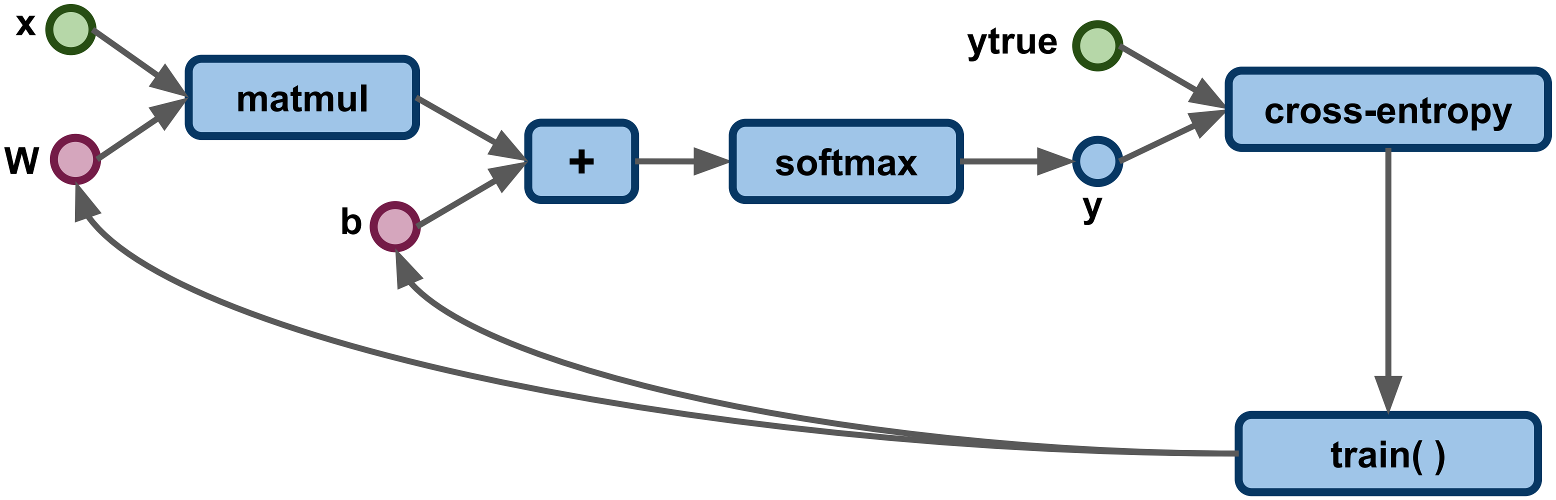
```
def train(model, images, labels, learning_rate):

    with tf.GradientTape() as t:
        current_loss = crossent_loss(labels, model(images))

    dW, db = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub(learning_rate * dW)
    model.b.assign_sub(learning_rate * db)
```

```
def training_loop(model, x, y, epochs=1):
    for e in range(epochs):
        ds = tf.data.Dataset.from_tensor_slices( (x, y) )
        ds = ds.shuffle( x.shape[0] ).batch(100)
        for (xbatch,ybatch) in ds:
            train(model, xbatch, ybatch, learning_rate=0.5)
    predicted = model(x)
    train_loss = crossent_loss(y, predicted)
    train_acc = accuracy( y, predicted )
    print("Final state: train_loss=%2.5f, train_acc=%2.5f" %
                 (train_loss, train_acc) )
```

Print out the model accuracy and the loss, evaluated on the input data.

Running the Computational Graph

Here's the graph we've built, so far:



Note: this is a simplification of the graph that TF would build for you. You can view the actual graph using TensorBoard: https://www.tensorflow.org/tensorboard/graphs

Assessing the model: test data

Once we've trained a model, how do we tell if it's good?

Use train/test split

- Data set aside ahead of time, which the model hasn't seen before
- Train on one set of data (train data), evaluate on another (test data)

What fraction of the labels did we get right?

```
def accuracy( y_true, y_pred ):
    hits = tf.math.equal( tf.math.argmax(y_true,1), tf.math.argmax(y_pred,1) )
    hits = tf.cast( hits, dtype=tf.float32 )
    return tf.reduce_mean( hits )
```

To "undo" the one-hot encoding, we take the argmax.

Putting it all together

TF includes the MNIST data. We just need to load it, rescale it and flatten the images using `tf.reshape`

```
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
```

TF MNIST pixels are integers 0 to 255. Rescale to be in [0,1]

```
train_shape = x_train.shape
x_train = tf.reshape( x_train, [train_shape[0], 28*28] )
test_shape = x_test.shape
x_test = tf.reshape( x_test, [test_shape[0], 28*28] )
```

Reshape the data so that each 28-by-28 image is now a 784-dimensional vector.

```
classifier =  SoftmaxModel( 784, 10 )
training_loop( classifier, x_train, y_train, 20 )
```

```
Final state: train_loss=0.29206, train_acc=0.92160
```

```
accuracy( y_test, classifier(x_test))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.913>
```

Putting it all together

```
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
```

```
train_shape = x_train.shape
x_train = tf.reshape( x_train, [train_shape[0], 28*28] )
test_shape = x_test.shape
x_test = tf.reshape( x_test, [test_shape[0], 28*28] )
```

```
classifier =  SoftmaxModel( 784, 10 )
training_loop( classifier, x_train, y_train, 20 )
```

```
Final state: train_loss=0.29206, train_acc=0.92160
```

Initialize the softmax classifier and fit it to the training data.

```
accuracy( y_test, classifier(x_test))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.913>
```

Now we're using the **test data** instead of the training data.

Putting it all together

```
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
```

```
train_shape = x_train.shape
x_train = tf.reshape( x_train, [train_shape[0], 28*28] )
test_shape = x_test.shape
x_test = tf.reshape( x_test, [test_shape[0], 28*28] )
```

```
classifier =  SoftmaxModel( 784, 10 )
training_loop( classifier, x_train, y_train, 20 )
```

```
Final state: train_loss=0.29206 train_acc=0.92160
```

```
accuracy( y_test, classifier(x_test))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.913>
```

Accuracy on test data is a bit worse than train. This is normal. We fit the model to the train data. On the other hand, the model has never seen the test data before.

Workshop II: Better Digit Recognition with NNs

Can we do better than 92% accuracy?

One obvious flaw:

Our softmax regression doesn't use structure of the image

How we vectorized our image didn't matter!

Two options:

1) Write down a better model
2) Use a neural net!

Crash Course: Neural Nets

Biologically-inspired computing model

Inputs processed by units ("neurons")

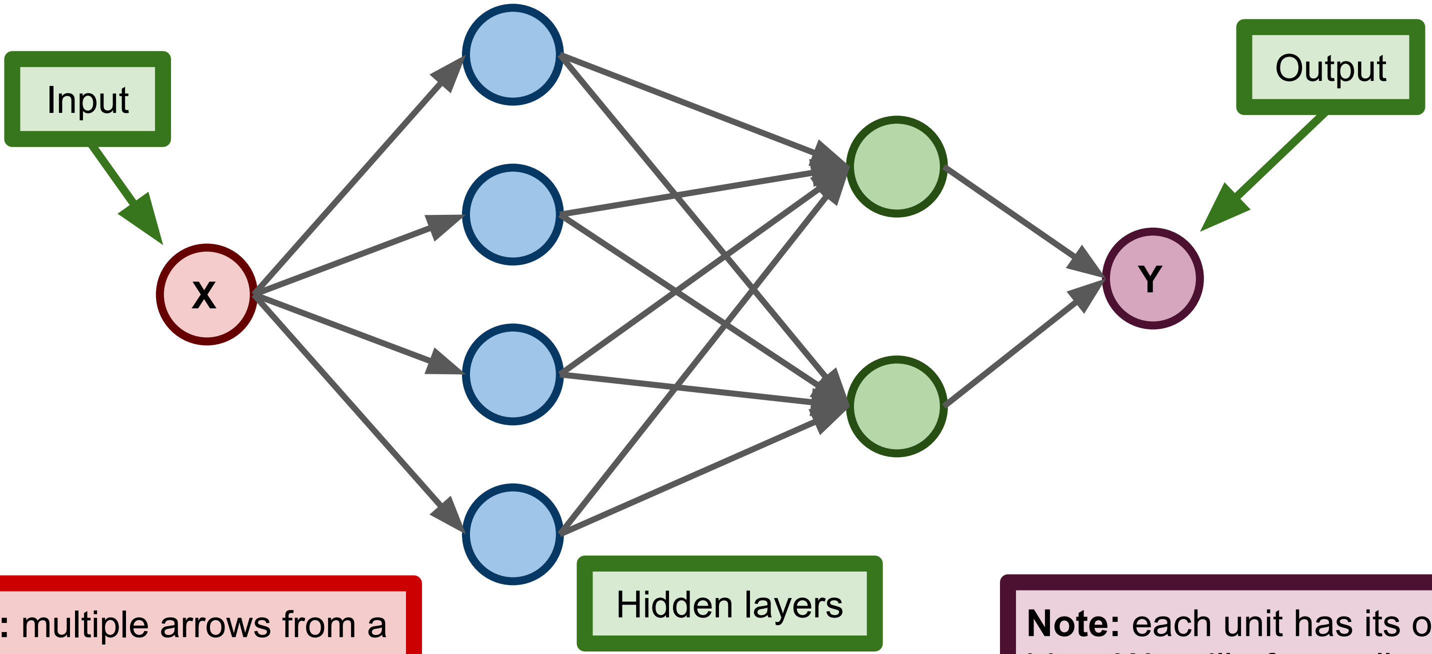
- Each unit outputs a function of some inputs
- Units apply linear functions to their inputs...
- ...followed by a nonlinear **activation function**

$$\overbrace{f(x)}^{\text{Output of unit}} = \overbrace{K}^{\text{Activation function}}\left(\sum_i \underbrace{w_i}_{\text{Weights}} \underbrace{g_i(x)}_{\text{Inputs to unit}}\right)$$

Goal: build a model that approximates some function

- **Ex:** input is an audio signal, output is a (prob. dist. over) word label
- **Ex:** input is English text, output is (prob. dist. over) French text
- **Ex:** input is an image, output is (prob. dist. over) label

Crash Course: Neural Nets



Note: multiple arrows from a unit denote **broadcast**, not different outputs.

Note: each unit has its own weight and bias. We will often collect the weights and biases from a single layer into a single tensor or pair of tensors.

Crash Course: Neural Nets

Early NNs: perceptron (Rosenblatt, 1957)

- Single-layer of computation
- Can only learn linearly separable functions
- https://en.wikipedia.org/wiki/Perceptron

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Multilayer perceptron (MLP)

- Multiple layers of units, can learn more complicated functions (e.g., XOR)
- https://en.wikipedia.org/wiki/Multilayer_perceptron

Feed-forward vs recurrent neural net (RNN)

- Feed-forward network is an acyclic graph
- RNN can have units whose outputs feed back to earlier units

Convolutional Neural Nets (CNNs)

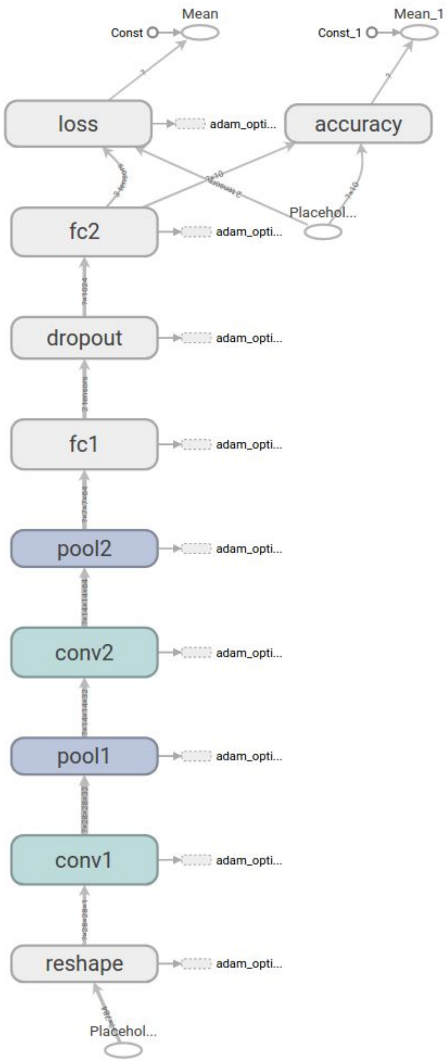
Deep (many layers)

Feed-forward (NN connections are acyclic)

Three basic types of layers:

- Convolutional
- Pooling
- Fully connected

Dropout “layer” provides regularization



Convolution

(Based on) an operation from signal processing

Roughly speaking, convolution computes response of a system to an input

https://en.wikipedia.org/wiki/Convolution

Typical NNs: units apply matrix multiplication followed by nonlinearity

CNN: units apply convolution instead of matrix multiplication

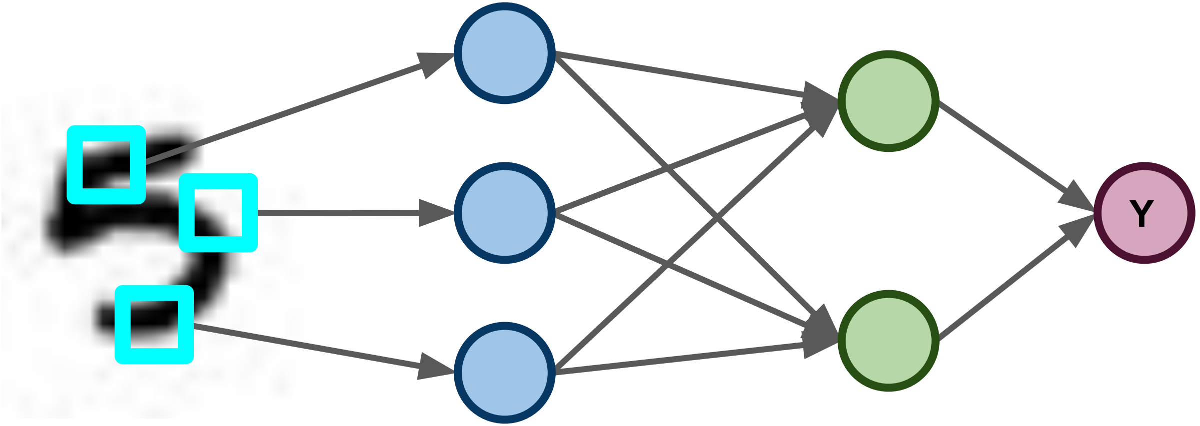
Still a linear operation

In image processing, units apply convolution to their **receptive fields**

Biologically inspired: e.g., neurons in visual cortex respond selectively

https://en.wikipedia.org/wiki/Receptive_field

Convolution: receptive fields



In image processing, units apply convolution to their **receptive fields**

Biologically inspired: e.g., neurons in visual cortex respond selectively

https://en.wikipedia.org/wiki/Receptive_field

Pooling

Typical setup: pass output of one unit to next layer

Pooling replaces this with a **summary statistic**

- Input to next layer is a function of several units from previous layer
- Example: pool adjacent pixels in an image

Common pooling operations:

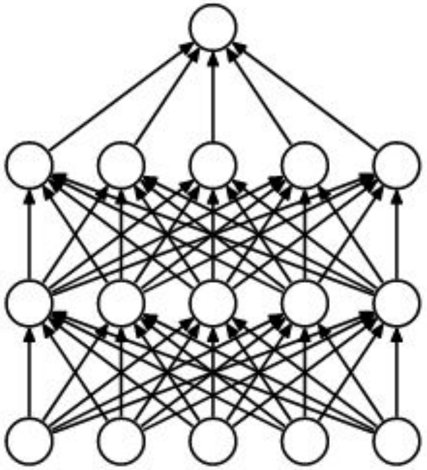
- Max pooling: report maximum value over the outputs
- (weighted) average: take weighted average over the outputs
  - Weighted according to, e.g., distance from center of receptive field

Dropout

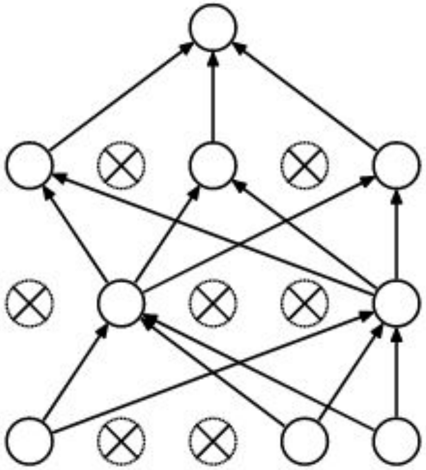
Common technique for regularization (avoiding overfitting)

At each training step, randomly choose some units to drop

- These units do not contribute to the network computation
- Forces other weights to “compensate”, introduces redundancy across units



(a) Standard Neural Net



(b) After applying dropout.

Image credit: Srivastava, et al (2014)
This is the paper in which dropout was initially suggested.
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

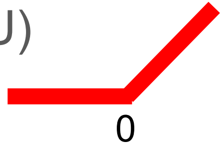
Building the Neural Net

Four layers

- Two convolutional layers
- Two fully-connected layers
- Dropout between FC layers

Nonlinearity: We'll use Rectified Linear Unit (RELU)

https://en.wikipedia.org/wiki/Rectifier_(neural_networks)



Pooling: max-pooling over 2-by-2 squares

Jupyter notebook:
http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2022/STAT679/democode/cnn_mnist_demo.ipynb