Statistics and Artificial Intelligence

Lecture 5: Computational Graphs and TensorFlow

Yixin Wang

Goals for Today

comp graph
→ generalizable way to reduce calculations to a unified operation

→ automatically return gradients we need

- Computation Graph
- Deep Learning Frameworks
- First Steps with Tensorflow

→ Frameworks preimplement many algorithms + programs

Suggestions and Questions from JiTTs

- Solutions to JiTTs available after due date
- Weekly FAQ post for points of confusion on piazza
  - We will address some of these in class too
- Materials are abstract; lectures are too fast— we'll try to be more concrete and slower.
  - e.g. We'll see computational graph on logistic regression today.
- Some of you had trouble finding my office hours
  - Tue 5-6pm in 445B WH
  - Wed 7:30-9:30pm on zoom (please check course calendar for details)

Computation Graph

- The computations of a neural network are organized in terms of
  - a forward pass or a forward propagation step, in which we compute the output of the neural network,
  - followed by a backward pass or back propagation step, which we use to compute gradients or compute derivatives.
- The computation graph explains why it is organized this way.

complicated to calculate the derivative of cost function

→ computational graph simplifies it

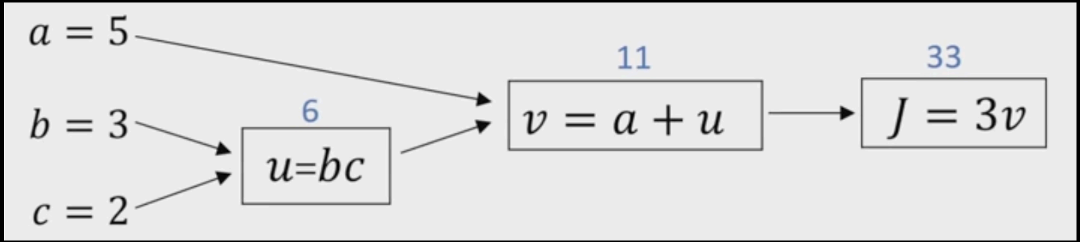
Computational Graph

Left-to-right pass

J(A, B, C) = 3(a + bc)

- Need to breakdown so a computer can do it
- use intermediate quantities so we only need to deal w/ 2 numbers at a time

- $J(a,b,c) = 3(a + bc)$
- Three steps: $u = bc$, $v = a+u$, $J=3v$



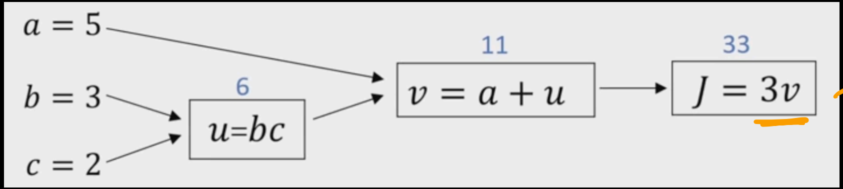
Forward Pass = Forward Propogation

Computing Derivatives with a Computation Graph

Right-to-left pass (Chain rule) $a \to v \to J$

$J(a,b,c) = 3(a+bc)$

$\frac{\partial J}{\partial v}$ If we nudge v, what is the resulting shift in J



$\frac{\partial J}{\partial v} = 3$

$\frac{\partial J}{\partial a} = J(v(a))$

$= \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a}$

$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial b}$

- $\frac{\mathrm{d}J}{\mathrm{d}v} = ?$
- Backpropagation
- $\frac{\mathrm{d}J}{\mathrm{d}a} = \frac{\mathrm{d}J}{\mathrm{d}v}\frac{\mathrm{d}v}{\mathrm{d}a}$, Propagate the change in a to v to J.

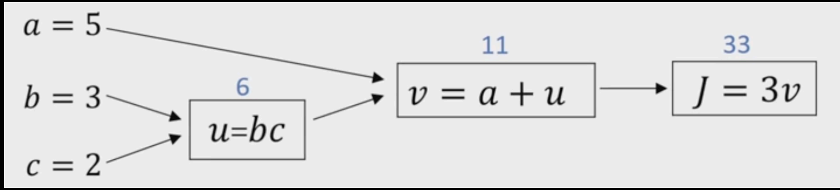
comp graph can help us identify shared components in computation

Computing Derivatives with a Computation Graph

Right-to-left pass (Chain rule) $a \rightarrow v \rightarrow J$

when training neural net we want to optimize loss or cost function

- Always w/ respect to same variable



notation

$$\frac{d\,\text{Final output}}{d\,\text{var}} = dJdvar$$

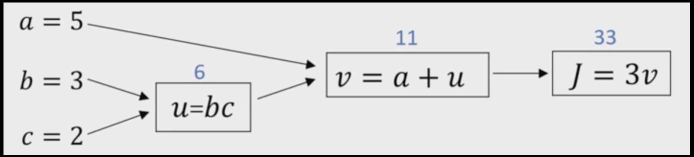
usually cost function

$$\frac{\partial J}{\partial a} \rightarrow dJda \rightarrow da$$

- $\dfrac{\mathrm{d}J}{\mathrm{d}v} = ?$
- Backpropagation
- $\dfrac{\mathrm{d}J}{\mathrm{d}a} = \dfrac{\mathrm{d}J}{\mathrm{d}v}\dfrac{\mathrm{d}v}{\mathrm{d}a}$, Propagate the change in a to v to J.

Computing Derivatives with a Computation Graph

Right-to-left pass (Chain rule) $b \rightarrow J, c \rightarrow J$



Don't need to redo calculations if yours go through shared paths

$J \rightarrow v \rightarrow u$ backwards

shared component

$J \rightarrow v \rightarrow u$ -▷ Backwards

$J \rightarrow v \rightarrow u \rightarrow c$

- $\frac{dJ}{du} = \frac{\partial J}{\partial v} \cdot \frac{dv}{du} = 3$
- $\frac{dJ}{db} \; \underbrace{\frac{\partial J}{\partial v} \cdot \frac{dv}{du}}_{3} \cdot \underbrace{\frac{du}{db}}_{2} = 6$
- $\frac{dJ}{dc} \; \frac{\partial J}{\partial v} \cdot \frac{du}{dc} = 3 \cdot 3 = 9$

Poll

- One step of ________ propagation on a computation graph yields derivative of final output variable.

- Backward

- Forward

Part of backward pass is to first look at forward passes

- use dependencies to find shared paths

Logistic Regression Gradient Descent

Logistic regression recap

- $z = \sum_{j}^{n_x} w_j x_j + b = \underline{w}^\top x + \underline{b}$

weighted sum · sigmoid

- $\hat{y} = a = \sigma(z)$
- $L(a, y) = -(y \log(a) + (1 - y)\log(1 - a))$

cross entropy loss

loss function: How far predictions are from true value

- The computational graph of logistic regression

$n_x = 2$ x_1, w_1, x_2, w_2, b

$z = w_1 x_1 + w_2 x_2 + b \rightarrow \hat{y} = a = \sigma(z)$

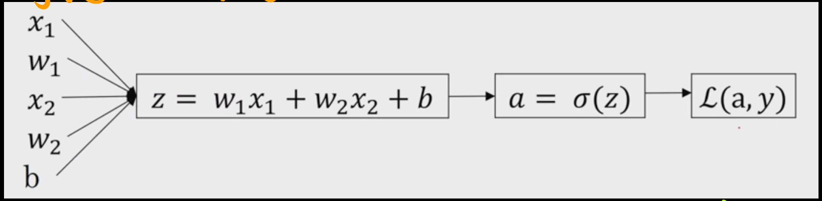
$\rightarrow L(a, y)$

special case of 2 dimensional inputs

Logistic Regression Gradient Descent

Logistic regression derivatives

use computational graph to calculate gradient



dz • $\frac{\partial L}{\partial z} = \frac{dL(a,y)}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)(a(1-a))$ L → A → z

dw_1 • $\frac{\partial L}{\partial w_1}$ $dw_1 = \frac{dL}{dz} \cdot \frac{dz}{dw_1} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)(a(1-a)) \cdot x_1$

dw_2 • $\frac{\partial L}{\partial w_2}$ $x_2 \cdot d_z$

db • $\frac{\partial L}{\partial b}$ $dz \cdot \underbrace{\frac{dz}{db}}_{\text{linear to 1}} = dz$

Gradient Descent

$w_1 \leftarrow w_1 - \alpha dw_1$ ← learning rate

$w_2 \leftarrow w_2 - \alpha dw_2$

$b \leftarrow b - \alpha db$

cost = average loss over many points

Logistic Regression Gradient Descent

Logistic regression on m examples

prediction

true label

- $$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$$
- $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^\top x^{(i)} + b)$
- Previously, we computed the gradient with one training example $(x^{(i)}, y^{(i)})$
- $$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1}}_{\text{calculate gradient w/ respect to parameter for every loss}} L(a^{(i)}, y^{(i)})$$

whole pipeline

Logistic Regression Gradient Descent

Logistic regression on m examples: The full algorithm

initialize all to 0

For each datapoint we go through forward pass

- $J = 0; dw_1 = 0; dw_2 = 0; db = 0;$
- For $i = 1$ to m
  - $z^{(i)} = w^\top x^{(i)} + b$
  - $a^{(i)} = \sigma(z^{(i)})$
  - $J += -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$
  - $dz^{(i)} = a^{(i)} - y^{(i)}$
  - $dw_1 += x_1^{(i)} dz^{(i)}$
  - $dw_2 += x_2^{(i)} dz^{(i)}$
  - $db += dz^{(i)}$
- $J /= m; \quad dw_1 /= m; \quad dw_2 /= m; \quad db /= m$

→ Backward pass, calculate derivatives

Do calculation for every data point
→ add up gradient, then average it

Logistic Regression Gradient Descent

Logistic regression on m examples: The full algorithm

- $J = 0; dw_1 = 0; dw_2 = 0; db = 0;$
- For $i = 1$ to m
    - $z^{(i)} = w^\top x^{(i)} + b$
    - $a^{(i)} = \sigma(z^{(i)})$
    - $J += -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$
    - $dz^{(i)} = a^{(i)} - y^{(i)}$
    - $dw_1 += x_1^{(i)} dz^{(i)}$
    - $dw_2 += x_2^{(i)} dz^{(i)}$
    - $db += dz^{(i)}$
- $J /= m; \quad dw_1 /= m; \quad dw_2 /= m; \quad db /= m$

$dw = \frac{dJ}{dw_1}$

Gradient Descent

✓ gradient

$w_1 \leftarrow w_1 - \alpha dw_1$

$w_2 \leftarrow w_2 - \alpha dw_2$

$b \leftarrow b - \alpha db$

learning rate

Poll

- In the for loop, why is there only one dw variable (i.e. no i superscripts in the for loop)?
- Only the derivative of one value is relevant.
- Only one derivative is being computed.
- The value of dw in the code is cumulative.

Questions?

Deep Learning Frameworks

Avoid reimplementing the same structure repeatedly

reuse code when possible

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Deep Learning Frameworks

Choosing deep learning frameworks

Don't want to reimplement every layer
-take as object + add together

- - Ease of programming (development and deployment)
- - Running speed computational graphs reduce calculation run time
- - Truly open (open source with good governance)

Faith it will remain open source

TensorFlow

Motivating Problem

don't just want to calculate function, also want to minimize

- Cost function: $J(w) = (w - 5)^2 = w^2 - 10w + 25$
- How can one implement it in TensorFlow to minimize this function?

File Edit View Insert Cell Kernel Widgets Help | Trusted | Python 3

In [1]:
```
import numpy as np
import tensorflow as tf
```

In [ ]:
```
w = tf.Variable(0, dtype=tf.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

cost = w ** 2 - 10 * w + 25
```

define w as TF variable

↳ by default optimizers try to minimize

Backpropagation with TensorFlow

```
In [1]: import numpy as np
        import tensorflow as tf

In [2]: w = tf.Variable(0, dtype=tf.float32)
        optimizer = tf.keras.optimizers.Adam(0.1)

        def train_step():
            with tf.GradientTape() as tape:
                cost = w ** 2 - 10 * w + 25
            trainable_variables = [w]
            grads = tape.gradient(cost, trainable_variables)
            optimizer.apply_gradients(zip(grads, trainable_variables))

        print(w)

        <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>

In [3]: train_step()
        print(w)

        <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.09999997>

In [4]: for i in range(1000):
            train_step()
        print(w)

        <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```

do every operation in forward + back pass

func only takes in w, optimize

input gradient w/ trainable variable

changes a little

resulting value close to its minimum

- Only need to implement the forward propagation (a sequence of operations) in Tensorflow.
    - Tensorflow can figure out how to perform backward propagation by itself via **gradient tape** (cf. Cassette tape)

Backpropagation with TensorFlow

Let the cost function depend on data

In [7]:
```
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])

    return w

w = training(x, w, optimizer)
print(w)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```

TensorFlow and Computation Graph

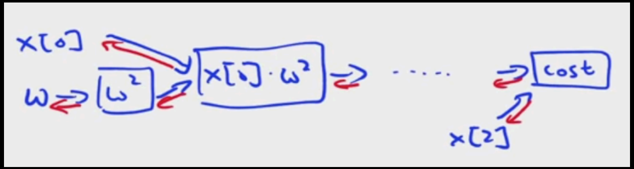
TensorFlow as equivalent object to computational graph

```
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])
    return w

w = training(x, w, optimizer)
```



First Steps with Tensorflow

- https://colab.research.google.com/drive/1VPE0xhheY-JlvlH9rEEkE26vi215obpE?usp=sharing

The materials in this course are adapted from materials created by Alexander Amini, Alfredo Canziani, Justin Johnson, Andrew Ng, Bhiksha Raj, Grant Sanderson and the 3blue1brown channel, Rita Singh, Ava Soleimany, and Ambuj Tewari.