15-442/15-642: Machine Learning Systems

Deep Learning and Programming Abstraction

Spring 2025

Tianqi Chen Carnegie Mellon University

1/15/2025

Outline

Overview of deep learning

Programming abstractions for deep learning

Outline

Overview of deep learning

Programming abstractions for deep learning

Elements of Machine Learning

Model(hypothesis) class

A parameterized function that describes how do we map inputs to predictions

- Loss function How "well" are we doing for a given set of parameters
- Training (optimization) method A procedure to find a set of parameters that minimizes the loss

$\hat{y}_i = \begin{bmatrix} \text{feature}_1 \\ \cdots \\ \text{feaure}_m \end{bmatrix} \qquad \hat{y}_i = \frac{1}{1 + \exp\left(-w^T x\right)}$

Logistic regression model

$$L(w) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \lambda ||w||^2$$

Regularized loss function

$$w \leftarrow w - \eta \nabla_w L(w)$$

Stochastic gradient descent

Deep Learning, Key Ideas

• Compositional multi-layer model



- End to end training: learning parameters of all layers together
- NOTE: the other ingredients (loss and training) remains the same as other machine learning methods

Understand Our Applications: An Overview of Deep Learning Models

- Convolutional Neural Networks
- Recurrent Neural Networks
- Transformers
- Graph Neural Networks
- Mixture-of-Experts

CNNs are widely used in vision tasks



Classification



Retrieval



ca pedestrians

Self-Driving



Detection



Synthesis

Segmentation

Convolution

• Convolve the filter with the image: slide over the image spatially and compute dot products



CNNs

• A sequence of convolutional layers, interspersed by pooling, normalization, and activation functions



Understand Our Applications: An Overview of Deep Learning Models

- Convolutional Neural Networks: vision tasks
- Recurrent Neural Networks
- Transformer
- Graph Neural Networks
- Mixture-of-Experts

one to one









Recurrent Neural Networks



How to Represent RNNs in Computation Graphs

- Computation graphs must be direct acyclic graphs (DAGs) but RNNs have self loops
- Solution: unrolling RNNs (define maximum depth)



When do we need RNNs?

- RNNs are designed to process sequences (texts, videos)
- RNNs are extremely useful when you want your model to have internal states when a sequence is processed
 - Commonly used in reinforcement learning (RL)

Understand Our Applications: An Overview of Deep Learning Models

- Convolutional Neural Networks
- Recurrent Neural Networks
- Transformers
- Graph Neural Networks
- Mixture-of-Experts

Inefficiency in RNNs?

- Problem: lack of parallelizability. Both forward and backward passes have O(sequence length) unparallelizable operators
 - A state cannot be computed before all previous states have been computed
 - Inhibits training on very long sequences



Attention: Enable Parallelism within a Sequence

 Idea: treat each position's representation as a query to access and incorporate information from a set of values



Attention: Enable Parallelism within a Sequence

- Idea: treat each position's representation as a query to access and incorporate information from a set of values
- Massively parallelizable: number of unparallelizable operations does not increase sequence length



We will learn attention and transformers in depth later:

- Self-attention
- Masked attention
- Multi-head attention

Understand Our Applications: An Overview of Deep Learning Models

- Convolutional Neural Networks
- Recurrent Neural Networks
- Transformers
- Graph Neural Networks
- Mixture-of-Experts

GNNs: Neural Networks on Relational Data

Classification

CAT

Single object



Graph Neural Network Architecture

Combine graph propagation w/ neural network operations



Understand Our Applications: An Overview of Deep Learning Models

- Convolutional Neural Networks
- Recurrent Neural Networks
- Transformers
- Graph Neural Networks
- Mixture-of-Experts

Mixture-of-Experts

 Key idea: make each expert focus on predicting the right answer for a subset of cases



Switch Transformers = Transformers + Mixture of Experts





Overview of deep learning

Programming abstractions for deep learning

Deep Learning Ingredients

- Model and architecture
- Objective function and training techniques
- Regularization, normalization and initialization (coupled with modeling)
 - Batch norm, dropout, Xavier
- Get good amount of data

Application affects System Design

Application

Data Management

Data Processing

System Design

Declarative language(SQL) Execution planner Storage engine Distributed Primitive(MapReduce) Fault tolerance layer Workload migration

Deep Learning Ingredients

- Model and architecture
- Objective function and training techniques
- Regularization, normalization and initialization (coupled with modeling)
 - Batch norm, dropout, Xavier
- Get good amount of data

Discussion how can these ingredients affect the system design of ML frameworks

Computational Graph Abstraction

- Nodes represents the computation (operation)
- Edge represents the data dependency between operations

Computational Graph for a * b +3



Case Study of Computational

- In the next few slides, we will do a case study of a deep learning program using TensorFlow v1 style API.
- Note that the most deep learning frameworks now use a different style, but share the same mechanism under the hood
- Think about abstraction and implementation when going through these examples

Logistic Regression





import tinyflow as tf from tinyflow.datasets import get mnist # Create the model x = tf.placeholder(tf.float32, [None, 784]) W = tf.Variable(tf.zeros([784, 10])) y = tf.nn.softmax(tf.matmul(x, W)) # Define loss and optimizer y_ = tf.placeholder(tf.float32, [None, 10]) cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1])) . `#-Update-pule---- $learning_rate = 0.5$ W_grad = tf.gradients(cross_entropy, [W])[0] train_step = tf.assign(W, W - learning_rate * W_grad) # Training Loop sess = tf.Session() sess.run(tf.initialize all variables()) mnist = get mnist(flatten=True, onehot=True) for i in range(1000): batch xs, batch ys = mnist.train.next batch(100) sess.run(train step, feed dict={x: batch xs, y :batch ys})

Loss function **Declaration**

$$P(\text{label} = k) = y_k$$
$$L(y) = \sum_{k=1}^{10} I(\text{label} = k) \log(y_i)$$

```
import tinyflow as tf
from tinyflow.datasets import get mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
                                                                                          Automatic Differentiation:
learning rate = 0.5
                                                                                          Next incoming topic
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize all variables())
mnist = get mnist(flatten=True, onehot=True)
for i in range(1000):
   batch xs, batch ys = mnist.train.next batch(100)
   sess.run(train step, feed dict={x: batch xs, y :batch ys})
```

```
import tinyflow as tf
from tinyflow.datasets import get mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
                                                                                               SGD update rule
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize all variables())
mnist = get mnist(flatten=True, onehot=True)
for i in range(1000):
   batch xs, batch ys = mnist.train.next batch(100)
   sess.run(train step, feed dict={x: batch xs, y :batch ys})
```

```
import tinyflow as tf
from tinyflow.datasets import get mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize all variables())
mnist = get mnist(flatten=True, onehot=True)
                                                                                              Real execution happens
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
                                                                                              here!
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

x = tf.placeholder(tf.float32, [None, 784])

- W = tf.Variable(tf.zeros([784, 10]))
- y = tf.nn.softmax(tf.matmul(x, W))



y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))



W_grad = tf.gradients(cross_entropy, [W])[0]

Automatic Differentiation, more details in follow up lectures



sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})



- What are the benefits of computational graph abstraction?
- What are possible implementation and optimizations on top of this abstraction



learning_rate

Imperative Computational Graph Construction

- TF1 style API uses a define then run approach
 - First construct the whole computational graph, then run the computation
- PyTorch and other frameworks uses a define and run approach
 - constructs the computational graph on the fly, along side the computations
- x = torch.Tensor([3])
 y = torch.Tensor([2])
 z = x y



Imperative Computational Graph Construction

- TF1 style API uses a define then run approach
 - First construct the whole computational graph, then run the computation
- PyTorch and other frameworks uses a define and run approach
 - constructs the computational graph on the fly, along side the computations

```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```



y.grad's path is omitted

TF1 vs PyTorch Style API

- Both leverages computational graph abstraction under the hood
- Define and run gives more flexibility to programmer

```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
print(z)
```

- Define then run still brings some benefits
 - See the entire computational graph to do global optimization
- Active topic of research, hybrid approaches such as JIT compilation