ONNX Training Proposal

AI Frameworks Team

Microsoft

Wei-Sheng Chin

Motivation

- Training can be a part of model deployment.
  - Reinforcement learning.
  - Transferring learning.
- User should be able to use whatever training framework/hardware they want to use to train a model.
  - Defining training locally and submitting it to a powerful cluster.
- To make AI exchangeable, ONNX should be expressive enough to cover training.

What is training?

- Given a data set, training is a procedure to improve a randomly initialized model by solving an optimization problem.
  - The trained model should fit the given data set better than its initial version.
- Here we focus on gradient-based methods due to its dominance in this field.

A Common Formulation to Train Neural Network

- Given some data points, we find an (sub-)optimal model by minimizing a quality measure.

$$\min_{w \in R^m} \Sigma_{i=1}^{l} L(x_i, w) - (1)$$

- w denotes all trainable model parameters.
- x_i is the i-th data point's feature vector.
- The measure, L, is a loss function, for example, squared distance in regression problem.

A Stochastic Gradient Method to Solve (1)

- $for\ t = 0, \dots,$
  - Randomly pick up $i \in \{1, \dots, l\}$
  - $G \leftarrow \nabla_w L(x_i, w)$
  - $w \leftarrow w - \eta G$
- Some observations
  - A loss function, L, we want to minimize.
  - Gradient is computed at each iteration based on the latest model w.
  - Gradient need to be manipulated before applying it.
  - The model w is updated by an assignment.
  - The pattern repeats at each iteration.

Fundamental Components to Support Training (# 2038)

- Loss function.
- Gradient computation.
- Optimizer for operators.
- Assignment semantic.
- Some spec changes to accommodate these new concepts.

Loss functions (PR #1939)

- We will support popular ones officially.
  - Mean squared error, $(y - \hat{y})^2$.
  - Absolute loss, $|y - \hat{y}|$.
  - Hinge loss, $\max(0, 1 - y\hat{y})$.
  - Logistic loss, $\log(1 - e^{-y\hat{y}})$.
  - Softmax cross entropy loss, $\log(\frac{e^{y_c}}{\sum e^{y_{c'}}})$.
- They should be composed as FunctionProto's by existing operators.

Gradient Operator (PR #2168)

- Attributes
  - xs: Names of the n differentiated tensors, for example, $["x_1", "x_2", \ldots, "x_n"]$.
  - y: Name of target tensor, for example, $"y"$.
- Inputs
  - Values of the source tensors. The i-th tensor in “Inputs” would be bound to the i-th name in “xs”.
- Outputs
  - $[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}]$.
- All outputs are optional. User can use empty string to indicate unnecessary ones.

Targeted Optimizers

- The number of Pytorch and TF optimizers is huge, so it’s not realistic to support all of them in the beginning.
- A list of our top priorities, selected by popularity and mathematical properties.
  - Momentum (PR # 1959)
  - Adagrad (PR #1955)
  - Adam (PR #1970)

Optimizer's Reference Implementation

- Each PR contains a numpy reference implementation.

```
def apply_adagrad(r, t, x, g, h, norm_coefficient, epsilon, decay_factor):
    # Compute adjusted learning-rate.
    r_ = r / (1 + t * decay_factor)
    # Add gradient of regularization term.
    g_regularized = norm_coefficient * x + g
    # Update squared accumulated gradient.
    h_new = h + g_regularized * g_regularized
    # Compute ADAGRAD's gradient scaling factors
    h_sqrt = np.sqrt(h_new) + epsilon
    # Apply ADAGRAD update rule.
    x_new = x - r_ * g_regularized / h_sqrt
    return (x_new, h_new)
```

Correctness of Optimizer Operators

- Reference implementation’s outputs are compared against both of Pytorch and Tensorflow in each PR.

Mix-precision Training

- Some recent researches show that some float32 parameters can be stored as float16.
- Low-precision numbers means less computation and less memory.
- ONNX optimizers should allow, for example, weights in float32 and gradient in float16.
  - It’s not difficult. We just allows more types in optimizers’ input list.

Storing Training Algorithm

- Training algorithm is broken into stages so that no assignment happens in each stage.
- One stage is just a computation graph (type: FunctionProto).
  - Gradient node + optimizer nodes + loss node + etc.
- Assignment is independently encoded outside stage's actual computation.
- Sequentially executing all stages is considered as one training iteration.

Storing Training Stages in ModelProto

- message ModelProto {

```
…
// The parameterized graph that is evaluated to execute the model.
optional GraphProto graph = 7;
// The functions can be referenced in the "graph.node" field and
// "training_info[i].algorithm.node" field.
repeated FunctionProto function = 9;
…
// Training-specific information. Sequentially executing all stored
// "TrainingInfoProto"s is one training iteration.
repeated TrainingInfoProto training_info = 21;
}
```

Storing Training Stage in TrainingInfoProto

- message TrainingInfoProto {

    // Training-specific initializers such as learning rate.
    repeated TensorProto initializer = 1;
    // Algorithm's inputs.
    repeated ValueInfoProto input = 2;
    // Selected outputs from "algorithms"
    repeated ValueInfoProto output = 3;
    // The training algorithm, accessing input, initializer, ModelProto.graph.initializer, ModelProto.graph.function.
    optional FunctionProto algorithm = 4;
    // String pair such as (“W_new”, “W”) to encode assignment semantic “$W = W\_new$”.
    repeated StringStringEntryProto update_binding = 7;

}

Summary Message on GitHub

Summary of Training Story in ONNX #2038

Open wschin opened this issue on May 21 · 2 comments

wschin commented on May 21 • edited

Member

This issue summarizes PRs related ONNX Training WG's progress.

We have a PR #2013 for defining required protobuf changes to enable training. That PR creates a new protobuf message to store `optimizer`, `loss`, and other useful fields. A graphical example can be found in ONNX Training Discussion.pptx

For loss, one can see #1939.
For distinguishing behaviors of operators in training and inference phases, see #1887.
For optimizers, we have #1955, #1970, #1959.
For gradient computation, #2168.

1

Thank you!