بسم الله الرحمن الرحیم

پروژه چهارم درس طراحی سخت افزار برای کاربرد های هوش مصنوعی

دکتر صالحی نسب

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

فهرست

تبدیل مدل پایه

ابتدا با کد زیر خروجی onnx می گیریم

```
dummy_input = torch.randn(1, 3, 32, 32, requires_grad=False)
dummy_input = dummy_input.to(device)
torch.onnx.export(model,                         # Your model
                  dummy_input,                   # Sample input
                  "/kaggle/working/base_model_out.onnx",                 #
Path to save the ONNX file
                  export_params=True,            # Store the trained
weights
                  opset_version=11,              # ONNX opset version (or
a version compatible with ONNX2C)
                  do_constant_folding=True,      # Execute constant
folding optimization
                  input_names = ['input_image'], # Custom name for the
input layer
                  output_names = ['predictions'],# Custom name for the
output layer
                  dynamic_axes={'input_image' : {0 : 'batch_size'}, #
Allow for dynamic batch_size
                                'predictions' : {0 : 'batch_size'}})
```

سپس پروژه onnx2c رو میسازیم

```
!mkdir build
%cd build
!cmake -DCMAKE_BUILD_TYPE=Release ..
!make onnx2c
```

حال onnx را به c تبدیل می کنیم

```
!./onnx2c /kaggle/working/base_model_out.onnx >
/kaggle/working/base_model_out.c
```

ارزیابی مدل پایه

از داده های تست خروجی می گیریم.

```
image_count = 0
with open("cifar10_test_images.bin", "wb") as img_file, \
```

```
      open("cifar10_test_labels.bin", "wb") as label_file:
      for i, (images, labels) in enumerate(testloader):
      image_data = images.squeeze(0).numpy().astype(np.float32)
image_data.tofile(img_file)

      label_data = np.array(labels.item()).astype(np.int32)
label_data.tofile(label_file)

      image_count += 1
      if image_count % 100 == 0:
            print(f"Processed {image_count} images.")

print(f"Saved {image_count} test images and labels.")
```

این کار داده های تست را حال داده های تست را روی مدل در c اجرا می کنیم.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#include "base_model_out.c"

#define IMAGE_BATCH 1
#define IMAGE_CHANNELS 3
#define IMAGE_HEIGHT 32
#define IMAGE_WIDTH 32
#define INPUT_SIZE (IMAGE_BATCH * IMAGE_CHANNELS * IMAGE_HEIGHT *
IMAGE_WIDTH)

#define NUM_CLASSES 10

int main() {
      FILE *image_file, *label_file;
      float (*input_data)[IMAGE_CHANNELS][IMAGE_HEIGHT][IMAGE_WIDTH] =
      (float (*)[IMAGE_CHANNELS][IMAGE_HEIGHT][IMAGE_WIDTH])
malloc(INPUT_SIZE * sizeof(float));

      float (*predictions)[NUM_CLASSES] =
      (float (*)[NUM_CLASSES]) malloc(NUM_CLASSES * sizeof(float));

      int32_t true_label;
      int correct_predictions = 0;
      int total_images = 0;

      if (!input_data || !predictions) {
```

```
      perror("Failed to allocate memory");
      return 1;
      }

      label_file = fopen("cifar10_test_labels.bin", "rb");

      if (!image_file || !label_file) {
      perror("Error opening data files. Make sure
'cifar10_test_images.bin' and 'cifar10_test_labels.bin' exist.");
      free(input_data);
      free(predictions);
      return 1;
      }

      printf("Starting model testing...\n");

      while (fread(input_data, sizeof(float), INPUT_SIZE, image_file) ==
INPUT_SIZE &&
            fread(&true_label, sizeof(int32_t), 1, label_file) == 1) {

      total_images++;

      entry((const float (*)[3][32][32])input_data, (float
(*)[10])predictions);

      int predicted_class = -1;
      float max_prob = -1.0f;
      for (int i = 0; i < NUM_CLASSES; ++i) {
            if (predictions[0][i] > max_prob) {
                  max_prob = predictions[0][i];
                  predicted_class = i;
            }
      }

      if (predicted_class == true_label) {
            correct_predictions++;
      }

      if (total_images % 100 == 0) {
            printf("Processed %d images. Current accuracy: %.2f%%\n",
                  total_images, (float)correct_predictions /
total_images * 100.0f);
      }
      }

      printf("\nTesting complete.\n");
```

```
    printf("Total images tested: %d\n", total_images);
    printf("Correct predictions: %d\n", correct_predictions);
    printf("Accuracy: %.2f%%\n", (float)correct_predictions /
total_images * 100.0f);

    fclose(image_file);
    fclose(label_file);
    free(input_data);
    free(predictions);

    return 0;
}
```

در نهایت خروجی ها را مقایسه می کنیم. توجه کنید که مدل ها در دو دفعه متفاوت آموزش دیده اند و با توجه به دقت نزدیک آنها به این معنی است که کار درست انجام شده. (اولی تست روی پایتون و دومی کد سی)

```
Epoch 15: Train Acc = 100.00%, Test Acc = 87.53%
```

```
Testing complete.
Total images tested: 10000
Correct predictions: 8744
Accuracy: 87.44%
```

تبدیل مدل های چندی سازی شده

تعریف گره سفارشی (Custom Node) برای onnx در PyTorch

برای این که گره های کوانتایز در خروجی onnx ظاهر شود باید برای آن این گره ها را تعریف کنیم و گرنه onnx گره ها را به مواردی که خود می داند می شکند و تبدیل می کند.
برای این کار در گره های سفارشی خود یعنی Round و ALSQPlus و WLSQPlus تعریف باید کنیم که onnx چطور با آنها رفتار کند. برای این کار باید متد symbolic را بنویسم. پس ابتدا ساختار symbolic را تعریف می کنیم.

متد symbolic

همانطور که گفته شد این متد برای onnx تعریف می کند که این گره چیست و با آن چطور رفتار کند. این متد که به صورت استاتیک تعریف می شود آرگومان ورودی اولیه آن g است که گراف محاسباتی onnx است که درون متد از آن استفاده می کنیم. سپس آرگومان های متد forward را قرار می دهیم.

در بدنه آن گره جدید گراف را تعریف می کنیم. این کار با صدا زدن متد op رو g انجام می شود. برای این کار در آرگومان ابتدایی دسته ای که آن گره در آن قرار می گیرد و خود گره را مشخص می کنیم به عنوان مثال lsqplus_ops::ALSQPlus سپس تنسور های ورودی گره را مشخص می کنیم که می توانیم از همان موارد آرگومان ورودی متد symbolic خود استفاده کنیم. در آخر هم اتریبیوت های آن گره را مشخص می کنیم که می تواند انواع مختلفی باشد و این موضوع را می توانیم از آخر نام آنها متوجه بشیم، این موارد به نوعی موارد ثابتی هستند که در گره وجود دارد. در نهایت باید مقدار بازگشتی بدهیم. این مقدار بازگشتی یا گره است یا واقعا مقداری است. این مقدار به گره بدی داده می شود. همچنین معمولا لازم است این مقدار بازگشتی با setType دقیقا مشخص شود.

تحلیل گره ALSQPlus

```
class ALSQPlus(Function):
      # ... forward and backward methods ...

      @staticmethod
      def symbolic(g, weight, alpha, beta, g_tensor, Qn, Qp):
      return g.op(f"{CUSTOM_OP_DOMAIN}::ALSQPlus",
                  weight, alpha, beta, g_tensor,
                  Qn_i=int(Qn),
                  Qp_i=int(Qp)).setType(weight.type())
```

مواردی که بالا گفتیم در این متد به این شکل می شود:

1. متد symbolic ورودی‌های g (گراف)، weight, alpha, beta, g_tensor (تنسورها) و Qn, Qp (مقادیر عددی) را دریافت می‌کند.
2. با استفاده از g.op() یک گره جدید در گراف ONNX ایجاد می‌کند.
3. نام گره: نام گره به صورت lsqplus_ops::ALSQPlus تعیین می‌شود. این کار باعث می‌شود گره شما کاملاً منحصربه‌فرد باشد.
4. ورودی‌های گره (Inputs): چهار ورودی اول (weight, alpha, beta, g_tensor) به عنوان ورودی‌های تنسوری گره به g.op داده می‌شوند.
5. صفات گره (Attributes): مقادیر Qn و Qp به عنوان Attribute به گره اضافه شده‌اند ((Qn_i=int(Qn)).

چرا این‌گونه تعریف شده؟

- Attributes به جای Inputs: متغیرهایی مانند Qn و Qp مقادیر ثابت و اسکالری هستند که رفتار کوانتیزه‌سازی را مشخص می‌کنند، اما خودشان تنسورهای بزرگی نیستند که در طول شبکه جریان داشته باشند. در استاندارد ONNX، بهترین راه برای ذخیره چنین پارامترهای ثابتی، استفاده از "Attribute" است. پسوند _i به صادرکننده می‌گوید که این یک Attribute از نوع integer است

(پسوندهای دیگر مانند f_ برای float و s_ برای string نیز وجود دارند). این کار مدل ONNX را بهینه‌تر و خواناتر می‌کند.

- setType: این بخش نوع داده خروجی گره را مشخص می‌کند (مثلاً float32 و با ابعاد مشابه weight). این اطلاعات برای گره‌های بعدی در گراف ضروری است.

تحلیل گره WLSQPlus

```
class WLSQPlus(Function):
      # ... forward and backward methods ...

      @staticmethod
      def symbolic(g, weight, alpha, g_tensor, Qn, Qp, per_channel):
      outputs = g.op(f"{CUSTOM_OP_DOMAIN}::WLSQPlus",
                  weight, alpha, g_tensor,
                  Qn_i=int(Qn),
                  Qp_i=int(Qp),
                  per_channel_i=int(per_channel),
                  outputs=2)

      outputs[0].setType(weight.type())
      outputs[1].setType(weight.type())
      return outputs
```

تمامی موارد مشابه قبل است تنها در اینجا چندین خروجی داریم. متد forward در کلاس WLSQPlus دو مقدار را برمی‌گرداند: w_q و wq.detach(). بنابراین، گره ONNX متناظر با آن نیز باید دو خروجی داشته باشد. آرگومان outputs=2 به g.op() می‌گوید که این گره دو خروجی تولید می‌کند.

گرفتن خروجی onnx

```
dummy_input = torch.randn(1, 3, 32, 32, requires_grad=False)
dummy_input = dummy_input.to(device)
torch.onnx.export(
            model,                  # مدل برای صدور
            dummy_input,            # ورودی نمونه
            "PATH/qX.onnx",         # مسیر فایل خروجی
            export_params=True,     # ذخیره وزن‌های آموزش‌دیده
            opset_version=11,       # ورژن استاندارد ONNX (مثلاً 11)
            do_constant_folding=True,# بهینه‌سازی برای حذف محاسبات ثابت
            input_names=['input'],   # نام ورودی
            output_names=['output'], # نام خروجی
            custom_opsets={         # **مهمترین بخش**
                  CUSTOM_OP_DOMAIN: CUSTOM_OPSET_VERSION # ثبت دامنه و ورژن
```

```
اپراتورهای سفارشی
            }
        )
```

کد نمایش داده شده، با استفاده از تابع torch.onnx.export، یک مدل PyTorch را به فرمت استاندارد ONNX تبدیل و ذخیره می‌کند. این تابع با دریافت یک ورودی نمونه (dummy_input)، گراف محاسباتی مدل را ردیابی کرده و ساختار آن را در فایل q8_new3.onnx می‌نویسد. پارامترهای دیگر مانند opset_version، نسخه اپراتورهای استاندارد ONNX را مشخص کرده، export_params باعث ذخیره وزن‌های آموزش‌دیده در فایل نهایی می‌شود و input_names و output_names نام‌های مشخصی را برای ورودی و خروجی مدل تعیین می‌کنند.

مهم‌ترین بخش در این کد، پارامتر custom_opsets است که برای مدیریت اپراتورهای سفارشی تعریف‌شده در کد (مانند ALSQPlus و WLSQPlus) حیاتی است. از آنجایی که ONNX این اپراتورها را به صورت پیش‌فرض نمی‌شناسد، این پارامتر دامنه سفارشی (CUSTOM_OP_DOMAIN) و ورژن آن را به صادرکننده معرفی می‌کند. این کار به صادرکننده اجازه می‌دهد تا گره‌های سفارشی را به درستی شناسایی کرده و یک فایل ONNX معتبر بسازد؛ در غیر این صورت، فرآیند صادرات به دلیل وجود اپراتورهای ناشناخته با خطا مواجه می‌شد.

اضافه کردن LSQPluse به onnx2c

حال که فایل onnx را داریم باید بتوانیم آن را به c تبدیل کنیم برای این کار لازم است گره های خود را برای onnx2c تعریف کنیم که برای این کار گام های زیر را طی می کنیم.

ایجاد گره ها

هر گره شامل ۳ بخش است:

- تعریف اتریبیوت ها (parseAttributes)
- پرینت خود (print)
- نحوه محاسبه خود (resolve)

برای اتریبیوت ها همان هایی که توی پایتون پاس دادیم را دریافت می کنیم.

```
    virtual void parseAttributes(onnx::NodeProto &node) override {
    for (const auto& a : node.attribute()) {
        LOG(TRACE) << "Parsing attribute " << a.name() << std::endl;
        if (a.name() == "Qn_i")
            Qn = parse_attribute_int(a);
        else if (a.name() == "Qp_i")
            Qp = parse_attribute_int(a);
```

برای محاسبه نیز ابتدا آرگومان ها را دریافت می کنیم و سپس تنسور خروجی را محاسبه می کنیم و بر می گردانیم.

```
      virtual void resolve(void) override {
      // ALSQPlus has 4 tensor inputs + 2 scalar attributes
      if (get_number_of_inputs() != 4)
            ERROR("ALSQPlus expects exactly 4 inputs: weight, alpha,
beta, g");

      name_input(0, "weight");
      name_input(1, "alpha");
      name_input(2, "beta");
      name_input(3, "g");

      const Tensor *weight = get_input_tensor(0);

      // Output has same shape and type as weight
      Tensor *output = new Tensor;
      output->data_dim = weight->data_dim;
      output->data_type = weight->data_type;
      register_output(output, "output");
      }
```

پرینت کردن چون کم اهمیت است به آن نمی پردازیم.

اضافه کردن گره های به graph.cc

برای این که گره های ما شناخته شوند و به عنوان کاندیدای گره های onnx مطرح شوند لازم است در این فایل اضافه شوند. ابتدا هدر های آنها را اضافه می کنیم.

```
#include "nodes/lsqplus_alsq.h"
#include "nodes/lsqplus_wlsq.h"
#include "nodes/lsqplus_round.h"
```

همچنین به متد createNode نیز باید برای این بررسی اضافه شوند که با ۲ شرط ساده آنها را اضافه می کنیم.

```
if (domain == "lsqplus_ops") {
      if (opName == "Round") return new LSQPlusRound;
      if (opName == "ALSQPlus") return new LSQPlusALSQ;
      if (opName == "WLSQPlus") return new LSQPlusWLSQ;
      ERROR("Unimplemented LSQPlus operator: " << opName);
}
```

همچنین مشخص است که فایل ها باید به cmakeList هم اضافه شوند.

کامپایل

برای کامپایل از کامند زیر استفاده می کنیم.

```
cmake -DCMAKE_BUILD_TYPE=Release .. && make onnx2c
```

نتایج تست

نکته: در تست ها داده های تست بین همه مدل های این فاز مشترک است اما داده ی تست می تواند با داده آموزشی همپوشانی داشته باشد زیرا در دفعه متفاوتی جدا شده. همچنین مدل پایه با یک آموزش دیگر نسبت به مدل های کوانتایز شده آموزش دیده.

برای تست یک برنامه ساده داریم که شبکه را می گیرد و کامپایل می شود و مواردی که مشاهده می کنید را خروجی می دهد.

```
--- Testing Model: ./model/q6.c ---

--- Results for Image 0 ---
  Logits: [-9.742605, -10.091438, -7.240261, +10.314564, -13.031570, -5.096244, -7.683680, -9.931599, -10.605585, -11.779073]
  Predicted Class: 3, True Label: 3

--- Results for Image 1 ---
  Logits: [+7.824708, +12.895650, -18.311428, -14.016900, -21.281385, -24.773722, -17.330744, -20.808477, +24.369431, -6.033176]
  Predicted Class: 8, True Label: 8

--- Results for Image 2 ---
  Logits: [-1.479768, +4.383913, -8.217711, -9.548512, -16.700287, -7.039883, -13.111302, -6.113712, +2.430023, -2.664546]
  Predicted Class: 1, True Label: 8

--------------------------------------------------
Total inference time for 3 images: 1.197316 seconds
--------------------------------------------------

Testing complete.
Total images tested: 10000
Correct predictions: 8744
Accuracy: 87.44%
```

```
--- Testing Model: ./model/q8.c ---

--- Results for Image 0 ---
  Logits: [-11.215549, -11.370474, -6.218935, +8.680490, -16.142017, -2.956226, -6.487892, -13.573064, -6.552884, -12.164732]
  Predicted Class: 3, True Label: 3

--- Results for Image 1 ---
  Logits: [+5.531449, +13.037809, -16.010151, -15.898207, -25.751812, -26.544092, -13.892951, -18.659245, +25.046473, -6.442247]
  Predicted Class: 8, True Label: 8

--- Results for Image 2 ---
  Logits: [-1.526087, +3.392154, -8.362632, -9.499461, -14.425511, -7.561287, -11.839763, -8.885445, +6.024962, -2.650465]
  Predicted Class: 8, True Label: 8

--------------------------------------------------
Total inference time for 3 images: 1.547844 seconds
--------------------------------------------------

Testing complete.
Total images tested: 10000
Correct predictions: 8706
Accuracy: 87.06%
```

```
--- Testing Model: ./model/q16.c ---

--- Results for Image 0 ---
  Logits: [+0.493005, +0.294757, +3.621235, +19.410875, -2.546337, +7.039279, +4.420062, -0.114820, +4.196938, -0.287317]
  Predicted Class: 3, True Label: 3

--- Results for Image 1 ---
  Logits: [+17.103575, +23.259344, -7.364806, -1.917630, -11.244974, -12.188890, -1.567738, -5.134895, +34.161358, +7.522946]
  Predicted Class: 8, True Label: 8

--- Results for Image 2 ---
  Logits: [+7.295977, +13.225143, -0.683340, +0.739433, -5.688441, +1.790246, +1.394970, +1.960757, +14.518518, +6.723339]
  Predicted Class: 8, True Label: 8

--------------------------------------------------
Total inference time for 3 images: 2.096619 seconds
--------------------------------------------------

Testing complete.
Total images tested: 10000
Correct predictions: 8735
Accuracy: 87.35%
```

```
--- Testing Model: ./model/base_model_out.c ---

--- Results for Image 0 ---
  Logits: [-7.115346, -7.075344, -9.867791, +11.963882, -11.620197, -0.274273, -5.448483, -9.865868, -6.366589, -8.034879]
  Predicted Class: 3, True Label: 3

--- Results for Image 1 ---
  Logits: [+3.673051, +9.443042, -11.923853, -11.426817, -18.502941, -13.861213, -10.878578, -12.668652, +18.046549, -0.837843]
  Predicted Class: 8, True Label: 8

--- Results for Image 2 ---
  Logits: [+0.477849, +4.638241, -5.695080, -8.775062, -16.083567, -7.796614, -14.365509, -5.741503, +8.489001, +3.975621]
  Predicted Class: 8, True Label: 8

--------------------------------------------------
Total inference time for 3 images: 2.024386 seconds
--------------------------------------------------

Testing complete.
Total images tested: 10000
Correct predictions: 8744
Accuracy: 87.44%
```

نتایج پروفایل

نتایج تست و پروفایل در پوشه results ذخیره شده که نتیجه پروفایل در همه مشابه هست و ۹۹ درصد زمان در توابع کانولوشن است. همچنین با توجه به زمان اجرا در یوزر و زمان واقعی اجرا برنامه CPU bounded هست و باید سرعت پردازش زیاد شود.

```
real  0m1.132s
user  0m1.130s
sys 0m0.002s
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.66      0.29     0.29        3    96.67    96.67  node__features_features_3_Conv
 24.78      0.57     0.28        3    93.33    93.33  node__features_features_10_Conv
 21.24      0.81     0.24        3    80.00    80.00  node__features_features_17_Conv
 14.16      0.97     0.16        3    53.33    53.33  node__features_features_7_Conv
 11.50      1.10     0.13        3    43.33    43.33  node__features_features_14_Conv
  1.77      1.12     0.02        3     6.67     6.67  node__features_features_0_Conv
```

برای کامپایل از فلگ Og برای بهبود سرعت و از pg برای پروفایل کردن استفاده شده.

مدل های زبانی استفاده شده در پروژه

https://github.com/copilot/share/c0220304-42a0-8483-a800-964484696135
https://g.co/gemini/share/0c39e1199fb0
https://g.co/gemini/share/b764da7e6583
https://g.co/gemini/share/c97c68d41f42
https://g.co/gemini/share/16dac1e44ad3
https://g.co/gemini/share/5f9d79f87854
https://g.co/gemini/share/28a22fefc4be
https://g.co/gemini/share/2c460347caf8
https://g.co/gemini/share/3d984f5ff1d6
https://g.co/gemini/share/20b87de32639
https://g.co/gemini/share/ced9c86a6d1e
https://g.co/gemini/share/9d4fae202acc
https://g.co/gemini/share/775872fb9aed
https://notebooklm.google.com/notebook/1e5e803c-88b5-4399-94c6-2f4b9b68aea1
https://notebooklm.google.com/notebook/d275dad6-2d66-4427-b536-4e199a8cc9cc