MNIST Classification with Orion

?

Orion is a dedicated Cairo-based library designed specifically to build machine learning models for ValidityML. Its purpose is to facilitate verifiable inference. For better performance we will operate with an 8-bit quantized model. In this tutorial, you will be guided on how to train your model using Quantized Aware Training using MNIST dataset, how to convert your pre-trained model to Cairo 1, and how to perform inference with Orion.

You can find all the code and the notebook in the dedicatedepository. Here is the content of the tutorial:

1. What is the MNIST dataset?
2. Train your model with Quantization-Aware Training
3. Convert your model to Cairo
4. Perform inference with Orion
5.

What is MNIST dataset?

The MNIST dataset is an extensive collection of handwritten digits, very popular in the field of image processing. Often, it's used as a reference point for machine learning algorithms. This dataset conveniently comes already partitioned into training and testing sets, a feature we'll delve into later in this tutorial.

The MNIST database comprises a collection of 70,000 images of handwritten digits, ranging from 0 to 9. Each image measures 28 x 28 pixels.

?

Train the model with Quantization-Aware Training

We will be usingTensorflow to train a neural network to recognize MNIST's handwritten digits in this tutorial. TensorFlow is a very popular framework for deep learning.

Dataset Preparation

In a notebook, import the required libraries and load the dataset.

```

Copy fromtensorflowimportkeras fromkeras.datasetsimportmnist fromscipy.ndimageimportzoom importnumpyasnp

(x_train,y_train),(x_test,y_test)=mnist.load_data()

```

We have a total of 70,000 grayscale images, each with a dimension of 28 x 28 pixels. 60,000 images are for training and the remaining 10,000 are for testing.

We now need to pre-process our data. For the purposes of this tutorial and performance, we'll resize the images to 14 x 14 pixels. You'll see later that the neural network's input layer supports a 1D tensor. We, therefore, need to flatten and normalize our data.

```

Copy

# Resizing function

defresize_images(images): returnnp.array([zoom(image,0.5)forimageinimages])

# Resize

x_train=resize_images(x_train) x_test=resize_images(x_test)

# Then reshape
```

x_train=x_train.reshape(60000,14*14) x_test=x_test.reshape(10000,14*14) x_train=x_train.astype('float32') x_test=x_test.astype('float32')

normalize to range [0, 1]

x_train/=255 x_test/=255

```

Model Definition and Training

We will design a straightforward feedforward neural network. Here's the model architecture we'll use:\

?

This model is composed of an input layer with a shape of 14*14, followed by two dense layers, each containing 10 neurons. The first dense layer uses a ReLU activation function, while the second employs a softmax activation function. Let's implement this architecture in the code.

```

Copy fromtensorflow.kerasimportlayers

num_classes=10

model=keras.Sequential([keras.layers.InputLayer(input_shape=(14*14,)), keras.layers.Dense(10, activation='relu'), keras.layers.Dense(10, activation='softmax')])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

```

Now let's train this model on our training data.

```

Copy batch_size=256 epochs=10 history=model.fit(x_train, y_train, epochs=epochs, validation_split=0.2)

```

```

Copy Epoch1/10 1500/1500[==============================]-1s 438us/step-loss:0.8641-accuracy:0.7506-val_loss:0.3855-val_accuracy:0.8947 Epoch2/10 1500/1500[==============================]-1s 391us/step-loss:0.3713-accuracy:0.8953-val_loss:0.3160-val_accuracy:0.9096 Epoch3/10 1500/1500[==============================]-1s 397us/step-loss:0.3252-accuracy:0.9070-val_loss:0.2916-val_accuracy:0.9150 Epoch4/10 1500/1500[==============================]-1s 389us/step-loss:0.3041-accuracy:0.9122-val_loss:0.2758-val_accuracy:0.9207 Epoch5/10 1500/1500[==============================]-1s 393us/step-loss:0.2917-accuracy:0.9153-val_loss:0.2672-val_accuracy:0.9237 Epoch6/10 1500/1500[==============================]-1s 386us/step-loss:0.2827-accuracy:0.9187-val_loss:0.2599-val_accuracy:0.9258 Epoch7/10 1500/1500[==============================]-1s 391us/step-loss:0.2752-accuracy:0.9201-val_loss:0.2554-val_accuracy:0.9273 Epoch8/10 1500/1500[==============================]-1s 390us/step-loss:0.2685-accuracy:0.9218-val_loss:0.2525-val_accuracy:0.9282 Epoch9/10 1500/1500[==============================]-1s 391us/step-loss:0.2635-accuracy:0.9235-val_loss:0.2491-val_accuracy:0.9303 Epoch10/10 1500/1500[==============================]-1s 392us/step-loss:0.2593-accuracy:0.9256-val_loss:0.2467-val_accuracy:0.9302

```

At this point, we have trained a regular model.

Making the model Quantization Aware

The aim of this tutorial is to guide you through the process of performing verifiable inference with the Orion library. As stated before, Orion exclusively performs inference on 8-bit quantized models. Typically, quantization is executed via two distinct methods: Quantization Aware Training (QAT) or Post-Training Quantization (PTQ), which occurs after the training phase. In this tutorial we will use QAT method.

Concretely QAT is a method where the quantization error is emulated during the training phase itself. In this process, the weights and activations of the model are quantized, and this information is used during both the forward and backward
```

passes of training. This allows the model to learn and adapt to the quantization error. It ensures that once the model is fully quantized post-training, it has already accounted for the effects of quantization, resulting in improved accuracy.

We will use TensorFlow Model Optimization Toolkit to finetune the pre-trained model for QAT.

```

Copy importtensorflow_model_optimizationastfmot

# Apply quantization to the layers

quantize_model=tfmot.quantization.keras.quantize_model

q_aware_model=quantize_model(model)

# 'quantize_model' requires a recompile

q_aware_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

q_aware_model.summary()

```

```

Copy Model: "sequential"

---

# Layer (type) Output Shape Param #

quantize_layer (QuantizeLa (None, 196) 3 yer)

quant_dense (QuantizeWrapp (None, 10) 1975 erV2)

quant_dense_1 (QuantizeWra (None, 10) 115 pperV2)

================================================================= Total params: 2093 (8.18 KB)
Trainable params: 2080 (8.12 KB) Non-trainable params: 13 (52.00 Byte)

---

```

We have now created a new model,q_aware_model , which is a quantization-aware version of our original model. Now we can train this model exactly like our original model.

```

Copy batch_size=256 epochs=10 history=q_aware_model.fit(x_train, y_train, epochs=epochs, validation_split=0.2)

scores,acc=q_aware_model.evaluate(x_test, y_test, verbose=0) print('Test loss:', scores) print('Test accuracy:', acc)

```

```

Copy Epoch1/10 1500/1500[==============================]-1s 563us/step-loss:0.2623-accuracy:0.9245-val_loss:0.2499-val_accuracy:0.9293 Epoch2/10 1500/1500[==============================]-1s 503us/step-loss:0.2539-accuracy:0.9260-val_loss:0.2473-val_accuracy:0.9296 Epoch3/10 1500/1500[==============================]-1s 508us/step-loss:0.2509-accuracy:0.9260-val_loss:0.2454-val_accuracy:0.9289 Epoch4/10 1500/1500[==============================]-1s 520us/step-loss:0.2484-accuracy:0.9278-val_loss:0.2444-val_accuracy:0.9293 Epoch5/10 1500/1500[==============================]-1s 533us/step-loss:0.2464-accuracy:0.9284-val_loss:0.2428-val_accuracy:0.9293 Epoch6/10 1500/1500[==============================]-1s 516us/step-loss:0.2440-accuracy:0.9282-val_loss:0.2409-val_accuracy:0.9309 Epoch7/10 1500/1500[==============================]-1s 540us/step-loss:0.2424-accuracy:0.9286-val_loss:0.2417-val_accuracy:0.9308 Epoch8/10 1500/1500[==============================]-1s 517us/step-loss:0.2409-accuracy:0.9294-val_loss:0.2391-val_accuracy:0.9304 Epoch9/10 1500/1500[==============================]-1s 539us/step-loss:0.2391-accuracy:0.9292-val_loss:0.2406-val_accuracy:0.9316 Epoch10/10 1500/1500[==============================]-1s 518us/step-loss:0.2380-
```

accuracy:0.9294-val_loss:0.2428-val_accuracy:0.9304 Test loss:0.246782124042511 Test accuracy:0.928600013256073

```

Converting to TFLite Format

Now, we will convert our model to TFLite format, which is a format optimized for on-device machine learning.

```

Copy importtensorflowastf

Create a converter

converter=tf.lite.TFLiteConverter.from_keras_model(q_aware_model)

Indicate that you want to perform default optimizations,

which include quantization

converter.optimizations=[tf.lite.Optimize.DEFAULT]

Define a generator function that provides your test data's numpy arrays

defrepresentative_data_gen(): foriinrange(500): yield[x_test[i:i+1]]

Use the generator function to guide the quantization process

converter.representative_dataset=representative_data_gen

Ensure that if any ops can't be quantized, the converter throws an error

converter.target_spec.supported_ops=[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

Set the input and output tensors to int8

converter.inference_input_type=tf.int8 converter.inference_output_type=tf.int8

Convert the model

tflite_model=converter.convert()

Save the model to disk

open("q_aware_model.tflite","wb").write(tflite_model)

```

Testing the Quantized Model

Now that we have trained a quantization-aware model and converted it to the TFLite format, we can perform inference using the TensorFlow Lite interpreter to test it.
```

We first load the TFLite model and allocate the required tensors. The Interpreter class provides methods for loading a model and running inferences.

```

Copy

# Load the TFLite model and allocate tensors.

interpreter=tf.lite.Interpreter(model_path="q_aware_model.tflite") interpreter.allocate_tensors()

```

Next, we get the details of the input and output tensors. Each tensor in a TensorFlow Lite model has a name, index, shape, data type, and quantization parameters. These can be accessed via the input_details and output_details methods.

```

Copy

# Get input and output tensors.

input_details=interpreter.get_input_details() output_details=interpreter.get_output_details()

```

Before performing the inference, we need to normalize the input to match the data type of our model's input tensor, which in our case is int8. Then, we use theset_tensor method to provide the input data to the model. We perform the inference using the invoke method.

```

Copy

# Normalize the input value to int8

input_shape=input_details[0]['shape'] input_data=np.array(x_test[0:1], dtype=np.int8) interpreter.set_tensor(input_details[0]['index'], input_data)

# Perform the inference

interpreter.invoke()

# Get the result

output_data=interpreter.get_tensor(output_details[0]['index']) print(output_data)

                                [[-128-128-6-6-128-116-128-128-128-128]]

```

Now, we are going to run the inference for the entire test set.

We normalize the entire test set and initialize an array to store the predictions.

```

Copy (,),(x_test_image,y_test_label)=mnist.load_data()

# Resize and Normalize x_test_image to int8

x_test_image=resize_images(x_test_image) x_test_image_norm=(x_test_image/255.0*255-128).astype(np.int8)
```

Initialize an array to store the predictions

predictions=[]

```

We then iterate over the test set, making predictions. For each image, we flatten the image, normalize it, and then expand its dimensions to match the shape of our model's input tensor.

```

Copy

Iterate over the test data and make predictions

foriinrange(len(x_test_image_norm)): test_image=np.expand_dims(x_test_image_norm[i].flatten(), axis=0)

Set the value for the input tensor

interpreter.set_tensor(input_details[0]['index'], test_image)

Run the inference

interpreter.invoke()

output=interpreter.get_tensor(output_details[0]['index']) predictions.append(output)

```

Finally, we use a function to plot the test images along with their predicted labels. This will give us a visual representation of how well our model is performing.

?

We have successfully trained a quantization-aware model, converted it to the TFLite format, and performed inference using the TensorFlow Lite interpreter.

Now let's convert the pre-trained model to Cairo, in order to perform verifiable inference with Orion library.

Convert your model to Cairo

In this section, you will generate Cairo files for each bias and weight of the model.

Create a new Scarb project

Scarb is a Cairo package manager. We will use Scarb to run inference with Orion. You can find all information about Scarb and Cairo installationhere .

Let's create a new Scarb project. In your terminal run:

```

Copy scarbnewmnist_nn

```

Replace the content in Scarb.toml file with the following code:

```

Copy [package] name="mnist_nn" version="0.1.0"

[dependencies] orion={ git="https://github.com/gizatechxyz/orion.git"}

[scripts] test="scarb cairo-test -f mnist_nn_test"

```
```

Finally, place the notebook andq_aware_model.tflite file in themnist_nn directory. We are now ready to generate Cairo files from the pre-trained model.

Generate Cairo files

In a new notebook's cell load TFLite and allocate tensors.

```

Copy

# Load the TFLite model and allocate tensors.

interpreter=tf.lite.Interpreter(model_path="q_aware_model.tflite") interpreter.allocate_tensors()

```

Then, create an object with an input from the dataset, and all weights and biases.

```

Copy

# Create an object with all tensors

# (an input + all weights and biases)

tensors={ "input":x_test_image[0].flatten(), "fc1_weights":interpreter.get_tensor(1), "fc1_bias":interpreter.get_tensor(2), "fc2_weights":interpreter.get_tensor(4), "fc2_bias":interpreter.get_tensor(5) }

```

Now let's generate Cairo files for each tensor in the object.

```

Copy

# Create the directory if it doesn't exist

os.makedirs('src/generated', exist_ok=True)

fortensor_name,tensorintensors.items(): withopen(os.path.join('src','generated',f"{tensor_name}.cairo"),"w")asf: f.write( "use core::array::ArrayTrait;\n"+ "use orion::operators::tensor::{TensorTrait, Tensor, I32Tensor};\n"+ "use orion::numbers::i32;\n\n"+ "\nfn{0}() -> Tensor ".format(tensor_name)+"{\n"+ " let mut shape = ArrayTrait::::new();\n" ) fordimintensor.shape: f.write(" shape.append({0});\n".format(dim)) f.write( " let mut data = ArrayTrait::::new();\n" ) forvalinnp.nditer(tensor.flatten()): f.write(" data.append(i32{{mag:{0}, sign:{1}}});\n".format(abs(int(val)),str(val<0).lower())) f.write( " TensorTrait::new(shape.span(), data.span())\n"+ "}\n" )

withopen(os.path.join('src','generated.cairo'),'w')asf: forparam_nameintensors.keys(): f.write(f"mod{param_name};\n")

```

Your Cairo files are generated insrc/generated directory.

Insrc/lib.cairo replace the content with the following code:

```

Copy modgenerated;

```

We have just created a file calledlib.cairo , which contains a module declaration referencing another module namedgenerated .

Let's analyze the generated files

Here is a file we generated:fc1_bias.cairo

```

Copy usecore::array::ArrayTrait; useorion::operators::tensor::{TensorTrait,Tensor,I32Tensor}; useorion::numbers::i32;

fnfc1_bias()->Tensor { letmutshape=ArrayTrait:::new(); shape.append(10); letmutdata=ArrayTrait:::new(); data.append(i32{ mag:1287, sign:false}); data.append(i32{ mag:3667, sign:true}); data.append(i32{ mag:2954, sign:false}); data.append(i32{ mag:7938, sign:false}); data.append(i32{ mag:3959, sign:false}); data.append(i32{ mag:5862, sign:true}); data.append(i32{ mag:4886, sign:false}); data.append(i32{ mag:4992, sign:false}); data.append(i32{ mag:10126, sign:false}); data.append(i32{ mag:2237, sign:true}); TensorTrait::new(shape.span(), data.span()) }

```

fc1_bias is ai32 Tensor two concepts that deserve a closer look.

Signed Integer in Orion

In Cairo, there are no built-in signed integers. However, in the field of machine learning, they are very useful. So Orion introduced a full implementation ofSigned Integer . It is represented by a struct containing both the magnitude and its sign as a boolean.

The magnitude represents the absolute value of the number, and the sign indicates whether the number is positive or negative.

```

Copy // Example of an i32. structi32{ mag:u32, sign:bool,// true means a negative sign. }

```

Tensor in Orion

The second concept Orion introduced is theTensor . We've used it extensively in previous sections, the tensor is a central object in machine learning. It is represented in Orion as a struct containing the tensor's shape, a flattened array of its data, and extra parameters. The generic Tensor is defined as follows:

```

Copy structTensor { shape:Span, data:Span }

```

You should now be able to understand the content in generated files.

Perform Inference with Orion

We have now reached the last part of our tutorial, performing ML inference in Cairo 1.0.

How to Build a Neural Network with Orion

In this subsection, we will reproduce the same model architecture defined earlier in the training phase with Tensorflow, but with Orion, as the aim is to perform the inference in Cairo.

Insrc folder, create ann.cairo file and reference the module inlib.cairo as follow:

```

Copy modgenerated; modnn;

```

Now, let's build the layers of our neural network innn.cairo . As a reminder, this was the architecture of the model we defined earlier:

Input -> FC1 (activation = 'relu') -> FC2 (activation= 'softmax') -> Output

Dense Layer 1

Innn.cairo let's create a functionfc1 that takes three parameters:

- i: Tensor

-
    - A tensor ofi32
- values representing the input data.
- w: Tensor
-
    - A tensor ofi32
- values representing the weights of the first layer.
- b: Tensor
-
    - A tensor ofi32
- values representing the biases of the first layer.
-

It should return aTensor .

```

Copy useorion::operators::tensor::core::Tensor; useorion::numbers::signed_integer::{integer_trait::IntegerTrait, i32::i32}; useorion::operators::nn::{NNTrait,I32NN};

fnfc1(i:Tensor, w:Tensor, b:Tensor)->Tensor { // ... }

```

To build the first layer, we need aLinear function and aReLU fromNNTrait .

```

Copy useorion::operators::tensor::core::Tensor; useorion::numbers::signed_integer::{integer_trait::IntegerTrait, i32::i32}; useorion::operators::nn::{NNTrait,I32NN};

fnfc1(i:Tensor, w:Tensor, b:Tensor)->Tensor { letx=NNTrait::linear(i, w, b); NNTrait::relu(@x) }

```

Dense Layer 2

In a similar way, we can build the second layerfc2 , which contains aLinear function and aSoftmax fromNNTrait . We could convert the tensor to fixed point in order to perform softmax, but for this simple tutorial it's not necessary.

```

Copy useorion::operators::tensor::core::Tensor; useorion::numbers::signed_integer::{integer_trait::IntegerTrait, i32::i32}; useorion::operators::nn::{NNTrait,I32NN};

fnfc1(i:Tensor, w:Tensor, b:Tensor)->Tensor { letx=NNTrait::linear(i, w, b); NNTrait::relu(@x) }

fnfc2(i:Tensor, w:Tensor, b:Tensor)->Tensor { NNTrait::linear(i, w, b) }

```

We are now ready to perform inference!

Make Prediction

Insrc folder, create atest.cairo file and reference the module inlib.cairo as follow:

```

Copy modgenerated; modnn; modtest;

```

In your test file, create a functionmnist_nn_test .

```

Copy

# [test]
```

[available_gas(9999999999999999)]

fnmnist_nn_test() { //... }

```

Now let's import and set the input data and the parameters generated previously.

```

Copy usemnist_nn::generated::input::input; usemnist_nn::generated::fc1_bias::fc1_bias;
usemnist_nn::generated::fc1_weights::fc1_weights; usemnist_nn::generated::fc2_bias::fc2_bias;
usemnist_nn::generated::fc2_weights::fc2_weights;

[test]

[available_gas(9999999999999999)]

fnmnist_nn_test() { letinput=input(); letfc1_bias=fc1_bias(); letfc1_weights=fc1_weights(); letfc2_bias=fc2_bias();
letfc2_weights=fc2_weights(); }

```

Then import and set the neural network we built just above.

```

Copy usemnist_nn::nn::fc1; usemnist_nn::nn::fc2; usemnist_nn::generated::input::input;
usemnist_nn::generated::fc1_bias::fc1_bias; usemnist_nn::generated::fc1_weights::fc1_weights;
usemnist_nn::generated::fc2_bias::fc2_bias; usemnist_nn::generated::fc2_weights::fc2_weights;

[test]

[available_gas(9999999999999999)]

fnmnist_nn_test() { letinput=input(); letfc1_bias=fc1_bias(); letfc1_weights=fc1_weights(); letfc2_bias=fc2_bias();
letfc2_weights=fc2_weights();

letx=fc1(input, fc1_weights, fc1_bias); letx=fc2(x, fc2_weights, fc2_bias); }

```

Finally, let's make a prediction. The input data represents the digit 7. So the index 7 should have the highest probability.

```

Copy usecore::array::SpanTrait;

usemnist_nn::nn::fc1; usemnist_nn::nn::fc2; usemnist_nn::generated::input::input;
usemnist_nn::generated::fc1_bias::fc1_bias; usemnist_nn::generated::fc1_weights::fc1_weights;
usemnist_nn::generated::fc2_bias::fc2_bias; usemnist_nn::generated::fc2_weights::fc2_weights;

useorion::operators::tensor::I32Tensor;

[test]

[available_gas(9999999999999999)]

fnmnist_nn_test() { letinput=input(); letfc1_bias=fc1_bias(); letfc1_weights=fc1_weights(); letfc2_bias=fc2_bias();
letfc2_weights=fc2_weights();

letx=fc1(input, fc1_weights, fc1_bias); letx=fc2(x, fc2_weights, fc2_bias);

letx=*x.argmax(0, Option::None(()),Option::None(())).data.at(0);

assert(x==7, 'shouldpredict7'); }

```

Test your model by runningscarb test .

```

Copy testingmnist_nn... running1tests testmnist_nn::test::mnist_nn_test...ok
testresult:ok.1passed;0failed;0ignored;0filteredout;

```

Bravo You can be proud of yourself! You just built your first Neural Network in Cairo 1.0 with Orion.

Orion leverages Cairo to guarantee the reliability of inference, providing developers with a user-friendly framework to build complex and verifiable machine learning models. We invite the community to join us in shaping a future where trustworthy AI becomes a reliable resource for all.

Join Discord Community Get Started Orion

Previous Tutorials Next Implement new operators in Orion

Last updated1 month ago
```