Build a Verifiable Neural Network with Giza Actions

Giza Actions provides developers with the tools to easily create and expand Verifiable Machine Learning solutions, transforming their Python scripts and ML models into robust, repeatable workflows. Models developed using the Action SDK possess a verifiable property, enabling you to encapsulate your model within a Zero-Knowledge cryptographic layer, thereby ensuring the integrity of the inference.

In this tutorial, we will explore the process of building your first Neural Network using MNIST dataset, Pytorch , and Giza Action SDK and demonstrating its verifiability.

You can directly follow the tutorial from within itsnotebook .

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. For the purpose of this tutorial, we will resize image to 14 x 14 pixels.

?

Login to Giza and create a Workspace

Before we begin, it's important to note that since we will be using Giza tools, you need to log in to the Giza platform. To do this, we recommend following these steps to install the Giza CLI, create a user, and generate API keys:

```

Copy pipxinstallgiza-cli gizauserscreate gizauserslogin gizauserscreate-api-key

```

For more detailed information aboutlogin , please refer to theGiza-CLI documentation .

If you haven't already created a workspace associated with your user, you'll need to do so. Workspaces in our platform are a crucial component designed to enhance user interaction with Giza Actions. These workspaces provide a user-friendly interface (UI) for managing and tracking runs, tasks, and metadata associated with action executions.

Please note that the workspace creation process can take up to 10 minutes as isolated resources are set up for each respective workspace. You can create a workspace using the following command:

```

Copy gizaworkspacescreate

```

If you've previously created a workspace with your account, you can retrieve your workspace URL as follows:

```

Copy gizaworkspacesget

```

For more detailed information aboutworkspaces , please refer to theWorkspace section of Giza-CLI documentation .

Define your model

To begin, we must define the architecture of our model. In this tutorial, we will construct a basic feedforward neural network

```

Copy importtorch importtorch.nnasnn importtorch.optimasoptim importtorchvision importnumpyasnp importlogging fromscipy.ndimageimportzoom fromgiza_actions.actionimportaction fromgiza_actions.taskimporttask fromtorch.utils.dataimportDataLoader,TensorDataset

device=torch.device('cuda'iftorch.cuda.is_available()else'cpu')

```

```

Copy input_size=196# 14x14 hidden_size=10 num_classes=10 num_epochs=10 batch_size=256 learning_rate=0.001

classNeuralNet(nn.Module): def__init__(self,input_size,hidden_size,num_classes): super(NeuralNet, self).**init**() self.input_size=input_size self.l1=nn.Linear(input_size, hidden_size) self.relu=nn.ReLU() self.l2=nn.Linear(hidden_size, num_classes)

defforward(self,x): out=self.l1(x) out=self.relu(out) out=self.l2(out) returnout
```

```

Prepare your Tasks

In the Giza Action SDK, a Task serves as a functional representation of a specific work segment within a Giza Actions workflow. Tasks offer a means to encapsulate portions of your workflow logic in traceable and reusable units across actions. Essentially, a Giza Actions task can accomplish nearly anything a typical Python function can. The unique aspect of tasks is their ability to access information about upstream dependencies and the status of these dependencies before starting. This capability enables tasks, for instance, to wait for the completion of another task before starting. Moreover, tasks benefit from Giza Actions' automated logging, which captures comprehensive details of each task run, including its duration, tags, and final state.

This is how we will encapsulate the methods we build using the@task decorator.

Prepare datasets and create loaders

We need to download datasets and create loarders for both training and testing purposes.

```

Copy defresize_images(images): returnnp.array([zoom(image[0], (0.5,0.5))forimageinimages])

@task(name=f'Prepare Datasets') defprepare_datasets(): print("Prepare dataset...") train_dataset=torchvision.datasets.MNIST(root='./data', train=True, download=True) test_dataset=torchvision.datasets.MNIST(root='./data', train=False)

x_train=resize_images(train_dataset) x_test=resize_images(test_dataset)

x_train=torch.tensor(x_train.reshape(-1,14*14).astype('float32')/255) y_train=torch.tensor([labelfor_, labelintrain_dataset], dtype=torch.long)

x_test=torch.tensor(x_test.reshape(-1,14*14).astype('float32')/255) y_test=torch.tensor([labelfor_, labelintest_dataset], dtype=torch.long)

print("✅ Datasets prepared successfully")

returnx_train,y_train,x_test,y_test

```

```

Copy @task(name=f'Create Loaders') defcreate_data_loaders(x_train,y_train,x_test,y_test): print("Create loaders...")

train_loader=DataLoader(TensorDataset(x_train, y_train), batch_size=batch_size, shuffle=True)
test_loader=DataLoader(TensorDataset(x_test, y_test), batch_size=batch_size, shuffle=False)

print("✅ Loaders created!")

returntrain_loader,test_loader

```

Train the model

We need to define our training method.

```

Copy @task(name=f'Train model') deftrain_model(train_loader): print("Train model...")

model=NeuralNet(input_size, hidden_size, num_classes).to(device) criterion=nn.CrossEntropyLoss()
optimizer=optim.Adam(model.parameters(), lr=learning_rate)

forepochinrange(num_epochs): fori,(images,labels)inenumerate(train_loader): images=images.to(device).reshape(-1,14*14)
labels=labels.to(device)

outputs=model(images) loss=criterion(outputs, labels)

optimizer.zero_grad() loss.backward() optimizer.step()

if(i+1)%100==0: print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss:{loss.item():.4f}')

print("✅ Model trained successfully") returnmodel

```

Test model

We need to define our testing method.

```

Copy @task(name=f'Test model') deftest_model(model,test_loader): print("Test model...") withtorch.no_grad(): n_correct=0 n_samples=0
forimages,labelsintest_loader: images=images.to(device).reshape(-1,14*14) labels=labels.to(device) outputs=model(images)
_,predicted=torch.max(outputs.data,1) n_samples+=labels.size(0) n_correct+=(predicted==labels).sum().item()
```
```

acc=100.0*n_correct/n_samples print(f'Accuracy of the network on the 10000 test images:{acc}%')

```

Action the tasks

Now that we've prepared the taks, we need to execute them. This will be done using theaction decorator.

Think of actions as unique types of functions. They are capable of receiving inputs, executing tasks, and producing outputs. Remarkably, transforming any standard function into a Giza Actions action is as simple as appending the@action decorator. This transformation alters the function's characteristics, bestowing several advantages:

- Each time this function is used, its activity is monitored, with every state change communicated to the API for efficient tracking of the action's execution.
- Input parameters undergo automatic type verification and adaptation to ensure they match the required types.
- In cases of failure, the system is equipped to retry. Time constraints can be imposed to avert excessively prolonged workflows.
- Actions leverage inherent logging capabilities, which record essential details of each action run, including its duration and conclusive state.
-

For more detailed information aboutactions , please refer to theActions section of Action SDK documentation.

```

Copy

@action(name=f'Execution', log_prints=True) defexecution(): x_train,y_train,x_test,y_test=prepare_datasets() train_loader,test_loader=create_data_loaders(x_train, y_train, x_test, y_test) model=train_model(train_loader) test_model(model, test_loader)

execution()

```

When you return to your workspace and refresh it, you will observe your action currently in the execution process.

To view all the actions you have run, navigate to the Action Runs section within your workspace. Each action run represents a single occurrence of executing the action.

?

Furthermore, by clicking on a particular action, you can closely monitor its progress. You'll have the ability to view logs, individual tasks, and associated metadata.

?

Giza Action enables you to deploy workflows, turning them from manual activation into API-managed entities that can be triggered remotely. Once a workflow is deployed, it creates a deployment within your Giza Workspace, which remains active, awaiting action runs associated with that deployment. When a run is detected, it is asynchronously executed in a local subprocess.

```

Copy @action(name=f'Execution', log_prints=True) defexecution(): x_train,y_train,x_test,y_test=prepare_datasets() train_loader,test_loader=create_data_loaders(x_train, y_train, x_test, y_test) model=train_model(train_loader) test_model(model, test_loader)

This can only be executed in a Python script, not in a notebook

if__name__=='**main**': action_deploy=Action(entrypoint=execution, name="pytorch-mnist-action") action_deploy.serve(name="pytorch-mnist-deployment")

```

All your deployments can be found in the 'Deployments' section of your Workspace.

?

Run and Prove!

Up to this point, we have primarily focused on training and testing our model using PyTorch while monitoring its execution through the Giza platform. However, if you are here, it's likely because you want to harness the capabilities of ZKML (Zero-Knowledge Machine Learning) and have the ability to demonstrate the integrity of your model's inferences.

In this section, we will delve into what it means to prove the integrity of inferences, what setup is required to make this possible, and how to verify the proof.

ZKML leverages validity proofs like SNARKs and STARKs, which enables the verification of the correctness of computational processes. By deploying such proof systems in machine learning applications, we gain the ability to validate the inference of ML models or to confirm that a specific input produced a certain output with a given model.
```

To generate ZK proofs for your model inferences, you must first convert your model into ZK circuits. This conversion process involves leveraging programming languages that specialize in building ZK circuits, such asCairo-lang . Subsequently, using the Giza-CLI, you can transpile your model from ONNX to Cairo. This process will be covered in the upcoming sections.

It's worth mentioning that at present, Orion and Action-SDK exclusively supports Cairo as a ZK backend. However, we are actively working on expanding support for other ZK backends (e.g; EZKL, Noir ...).

?

Convert to ONNX

Before invoking the Giza transpiler to convert your model into Cairo, you must first ensure that your model is converted to ONNX. We will explore this process in the following section.

ONNX, short for Open Neural Network Exchange, is an open format for representing and exchanging machine learning models between different frameworks and libraries. It serves as an intermediary format that allows you to move models seamlessly between various platforms and tools, facilitating interoperability and flexibility in the machine learning ecosystem.

```

Copy importtorch.onnx

@task(name=f'Convert To ONNX') defconvert_to_onnx(model,onnx_file_path): dummy_input=torch.randn(1, input_size).to(device) torch.onnx.export(model, dummy_input, onnx_file_path, export_params=True, opset_version=10, do_constant_folding=True)

print(f"Model has been converted to ONNX and saved as{onnx_file_path}")

@action(name="Action: Convert To ONNX", log_prints=True) defexecution(): x_train,y_train,x_test,y_test=prepare_datasets() train_loader,test_loader=create_data_loaders( x_train, y_train, x_test, y_test) model=train_model(train_loader) test_model(model, test_loader)

# Convert to ONNX

onnx_file_path="mnist_model.onnx" convert_to_onnx(model, onnx_file_path)

execution()

```

Transpile your model to Cairo and deploy on Giza

Follow these steps to transpile your model to Orion Cairo code, compile the transpiled project, and deploy it on the Giza platform.

Step 1: Transpile Your Model

Now that your model is converted to ONNX format, use the Giza-CLI to transpile it to Orion Cairo code:

```

Copy

giza transpile mnist_model.onnx --output-path verifiable_mnist

[giza][2024-02-07 16:32:13.511] Transpilation is fully compatible. Version compiled and Sierra is saved at Giza ✅

```

Step 2: Deploy Your Model

Thanks to full support for all operators used by MNIST model in the transpiler, your transpilation process is completely compatible. This ensures that your project compiles smoothly and has already been compiled behind the scenes on our platform.

If your model incorporates operators that aren't supported by the transpiler, you may need to refine your Cairo project to ensure successful compilation. For more details, refer to the tohow-to-guide . With your model transpiled and compiled, it's now ready for deployment on the Giza platform. Our deployment process sets up services that handle prediction requests via a designated endpoint, using Cairo to ensure the provability of inferences.

Creating a New Deployment Service

Deploy your service, which will be ready to accept prediction requests at the/cairo_run endpoint, by using the following command:

```

Copy gizadeploymentsdeploy--model-id--version-id

```

Run your program

Now that your Cairo model is deployed on the Giza platform, you have the capability to execute it. If you initiate a prediction using Giza Action without theverifiable mode, it runs the onnx version of the model.

When you initiate a prediction using Giza Action inverifiable mode, it executes the Sierra program using CairoVM, generating trace and memory files for the proving. It also returns the output value and initiates a proving job to generate a Stark proof of the inference.

First, let's make a prediction with the ONNX model (veriable=False)

```

Copy fromgiza_actions.modelimportGizaModel importtorch.nn.functionalasF

@task(name=f'Preprocess Image') defpreprocess_image(image_path): fromPILimportImage importnumpyasnp

# Load image, convert to grayscale, resize and normalize

image=Image.open(image_path).convert('L')

# Resize to match the input size of the model

image=image.resize((14,14)) image=np.array(image).astype('float32')/255 image=image.reshape(1,196)# Reshape to (1, 196) for model input returnimage

@task(name=f'Prediction with ONNX') defprediction(image): model=GizaModel(model_path="./mnist_model.onnx")

result=model.predict( input_feed={"onnx::Gemm_0": image}, verifiable=False )

# Convert result to a PyTorch tensor

result_tensor=torch.tensor(result)

# Apply softmax to convert to probabilities

probabilities=F.softmax(result_tensor, dim=1)

# Use argmax to get the predicted class

predicted_class=torch.argmax(probabilities, dim=1)

returnpredicted_class.item()

@action(name=f'Execution: Prediction with ONNX', log_prints=True) defexecution(): image=preprocess_image("./zero.jpg") predicted_digit=prediction(image) print(f"Predicted Digit:{predicted_digit}")

returnpredicted_digit

execution()

```

Now, let's make a prediction with the Cairo model (veriable=True).

Update the IDs in the following code with your own. ```

Copy MODEL_ID=199# Update with your model ID VERSION_ID=7# Update with your version ID

@task(name=f'Prediction with Cairo') defprediction(image,model_id,version_id): model=GizaModel(id=model_id, version=version_id)

(result,request_id)=model.predict(input_feed={"image": image}, verifiable=True)

Convert result to a PyTorch tensor

result_tensor=torch.tensor(result)

Apply softmax to convert to probabilities

probabilities=F.softmax(result_tensor, dim=1)

Use argmax to get the predicted class

predicted_class=torch.argmax(probabilities, dim=1)

returnpredicted_class.item(),request_id

@action(name=f'Execution: Prediction with Cairo', log_prints=True) defexecution(): image=preprocess_image("./imgs/zero.png") (result,request_id)=prediction(image, MODEL_ID, VERSION_ID) print("Result: ", result) print("Request id: ", request_id)

returnresult,request_id

execution()

```

Download and Verify the proof

When you ran your model, CairoVM generated trace and memory files under the hood to prove your program. We have initiated a proof job on your behalf and completed the proof.

Executing the following command, we will download the ZK proof generated in the proof-job.

```

Copy gizadeploymentsdownload-proof--model-id--version-id--deployment-id--proof-id--output-path

```

Now, let's verify the proof! All you need to do is simply running:

```

Copy gizaverify--proofPATH_OF_THE_PROOF

```

Voilà You've learned how to use the entire Giza stack, from training your model to transpiling it to Cairo execution. You've also learned how to verify the proof. We hope you've enjoyed this journey!

If you have any questions, please feel free to join us in ourDiscord , oropen an issue .

Previous GizaModel Next Token Trend Forecasting

Last updated1 day ago
```