

Checkpointing and Saving the States of AI Models

Ritu Arora, Venra Tech Inc.

1. Why is checkpointing useful in the field of machine learning and AI?

AI models that need to be trained with large datasets and for long durations can be made resilient to interruptions (such as network or server failures) and can be made to stop and resume the training as needed. Additionally, while training the AI models, it may be important in some situations to save and inspect the intermediate results produced during the model optimization process. Checkpointing is a technique that can be immensely useful in the aforementioned scenarios as it helps in saving the states of the models at regular intervals. By doing so, **it can help in saving the time and cost that may otherwise be wasted in the event of any interruption during the model training process.**

2. What happens during the process of checkpointing an AI model during training?

The process of checkpointing an AI model during training involves saving the information related to its state (such as model weights, epochs, loss, and model architecture) in a file. Then later, the state is read from the file (or "loaded") to resume the model training from the checkpoint. The developers can specify the frequency at which the checkpoints should be written along with the choice of whether the model state should be written to a separate file or should a previously saved state be overwritten with the latest information.

3. How does Tensorflow support checkpointing and saving model states?

Tensorflow provides separate APIs for (1) saving and loading the models and (2) for checkpointing. Let us review some of the differences in "checkpointing" a Tensorflow model and "saving" it. While the checkpoints in Tensorflow include the values of the trained parameters used by the models (e.g., model weights and biases), they do not include any description of the computations that are done with the models, and hence, the same source code that was used for writing a checkpoint and reading it back is required for using a checkpoint in future. In situations where **only model weights should be saved** (perhaps for the inference step), checkpointing is recommended.

In contrast to checkpointing, saving a model in Tensorflow includes a serialized description of the computations defined by the model along with the trained parameter values, thereby making the saved models independent of the original source code that was used to build them [1, 2].

The `tf.saved_model` API or the `tf.keras.Model` API can be used for saving models in Tensorflow, whereas the `tf.train.checkpoint` API in Tensorflow or `ModelCheckpoint` method in Keras can be used for checkpointing.

4. How are model states saved and loaded in Tensorflow?

There are two file formats that the developers can choose from while saving the model states: `SavedModel` and `HDF5`. Using the `HDF5` format results in a single file containing the saved state of the model, whereas using the `SavedModel` results in multiple files in a folder. Below are the code snippets that show how to save the model in the `HDF5` and `SavedModel` formats.

#In the lines below, "model" is a `tf.keras.Model` object

```
#below is how to save a model in the HDF5 format - a *.h5 file will be
#created on the disk
model.save("mytestmodel.h5")

#below is how to save a model in the SavedModel format
model.save("mytestmodel")
```

Upon saving the model in the SavedModel format a folder with the name mytestmodel will be created on the disk and will contain multiple files as shown below:

```

└─ mytestmodel
  ├── assets
  ├── variables
  ├── fingerprint.pb
  ├── keras_metadata.pb
  └── saved_model.pb

```

The saved_model.pb file in the mytestmodel folder shown above stores the Tensorflow program and the signatures of functions that accept tensor input or produce tensor output [3]. The fingerprint.pb file contains 64-bit hashes that uniquely identify the contents of the saved model. The assets directory stores the files used by the Tensorflow graph. The variables directory contains the values of the training parameters. The keras_metadata.pb file contains the metadata needed by Keras.

The SavedModel file format is used as default since Tensorflow 2.x and is recommended when a model has custom layers because during the loading step, the custom layers can be directly loaded without requiring any prior definition of those layers. However, with the HDF5 format, separate steps for defining the custom layers are required during the loading step.

In order to use the saved models for further actions such as resuming the training or doing predictions, the models would need to be loaded into to model object as shown below.

```
#loading a model that is saved in HDF5 format on disk
model = tf.keras.models.load_model("mytestmodel.h5")

#loading a model that is saved in the SavedModel format on disk
model = tf.keras.models.load_model("mytestmodel")
```

5. How are checkpoints written and loaded in Tensorflow?

TensorFlow supports the feature of checkpointing and restart by offering APIs for saving and loading the model states, thereby reducing the training time during the restart phase in the event of an interruption.

The ModelCheckpoint callback is used together with model.fit() to create a checkpoint file at a specific frequency [4, 5]. This callback (or a piece of code that is executed when certain conditions are met) allows specifying certain options such as filepath (it is a required parameter and it refers to the path at which the checkpoint file should be stored), frequency of checkpointing (in the example below it is set to "epoch" which means save the model after every epoch), monitor (that determines the model metric to

be monitored and in the example below the monitored metric is 'val_loss'), verbose (0 for silent mode and 1 for verbose mode for logging), save_best_only (if this is set to true, it will save when the model is considered the best according to the monitored metric which is 'val_loss' here), and mode (which is 'min' here because the 'val_loss' should be minimized).

```
#the code below will create a checkpoint
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filename,
monitor='val_loss', verbose=1, save_best_only=True, mode='min',
save_freq="epoch")

model.fit(x_train, y_train, epochs=5, batch_size = 100, validation_split =
0.1, callbacks=[checkpoint_callback])
```

The `load_model` call is used for loading the model state saved in the checkpoint file.

```
#the code will load the saved model/checkpoint from a file
model = tf.keras.models.load_model(filename)
```

The complete executable code demonstrating the use of the **ModelCheckpoint** callback for checkpointing and restart is shown in Listing 1. MNIST dataset is used with the sequential model in this example. The state of the model is saved in an HDF5 file named `testmodel12.h5`. The model is trained for 5 epochs with a batch size of 100. The total loss and accuracy of the model are calculated. A new instance of the model is created but is not trained. Next, the weights of the previously trained instance of the model that were saved in the `testmodel12.h5` file are loaded in this new instance of the model and the new instance is evaluated. When the new instance of the model is evaluated (after loading the weights from the checkpoint file), it shows the same values for the metrics as the trained model.

```
import os
import tensorflow as tf
from tensorflow import keras

mnist = tf.keras.datasets.mnist

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

#Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

filename = "testmodel12.h5"

#Check if checkpoint file exists. If it does, load the model and skip #building the
model.
```

```

if (os.path.isfile(filename)):
    print("Restarting")
    model = tf.keras.models.load_model(filename)
else:
    print('Building the model from beginning')
    model = tf.keras.models.Sequential([
        keras.Input(shape=input_shape),
        tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(num_classes, activation="softmax"),
    ])

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

checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filename,
monitor='val_loss', verbose=1, save_best_only=True, mode='min', save_freq="epoch")

model.fit(x_train, y_train, epochs=5, batch_size = 100, validation_split = 0.1,
callbacks=[checkpoint_callback])

totalScore=model.evaluate(x_test, y_test, verbose=2)
print("Test loss:", totalScore[0])
print("Test accuracy:", totalScore[1])

# Create a basic model instance again for comparison purposes
model = tf.keras.models.Sequential([
    keras.Input(shape=input_shape),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(num_classes, activation="softmax"),
])

model.compile(optimizer='adam',

```

```

        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

# Evaluate the recreated model
totalScore= model.evaluate(x_test, y_test, verbose=2)
print("Untrained model, accuracy: {:.2f}%".format(100 * totalScore[1]))
model.summary()

# Now loads the weights from the saved checkpoint
model.load_weights('testmodel2.h5')

# Re-evaluate the model and compare its output with the previous untrained #model
run
testScore = model.evaluate(x_test, y_test, verbose=2)

```

Listing 1. Code showing the use of ModelCheckpoint callback for checkpointing and restart.

The output associated with the steps for training and evaluating the model before checkpointing and during restart is shown in Listing 2.

Note: The model was rerun a few times before running the code from lines # 47-72 and its accuracy improved further during those runs.
Note: The model can be run interactively or in batch mode.

Output:
Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 [=====] - 1s 0us/step

Building the model from beginning
Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_2 (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense_4 (Dense)	(None, 10)	16010

=====
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0

Epoch 1/20
540/540 [=====] - ETA: 0s - loss: 0.3246 - accuracy: 0.9019
Epoch 1: val_loss improved from inf to 0.07674, saving model to testmodel2.h5
540/540 [=====] - 41s 74ms/step - loss: 0.3246 - accuracy: 0.9019 - val_loss: 0.0767 - val_accuracy: 0.9778
Epoch 2/20
540/540 [=====] - ETA: 0s - loss: 0.1020 - accuracy: 0.9688
Epoch 2: val_loss improved from 0.07674 to 0.05121, saving model to testmodel2.h5

```

540/540 [=====] - 39s 72ms/step - loss: 0.1020 - accuracy: 0.9688 -
val_loss: 0.0512 - val_accuracy: 0.9865
. . .

Epoch 20/20
540/540 [=====] - ETA: 0s - loss: 0.0251 - accuracy: 0.9914
Epoch 20: val_loss did not improve from 0.02863
540/540 [=====] - 39s 72ms/step - loss: 0.0251 - accuracy: 0.9914 -
val_loss: 0.0316 - val_accuracy: 0.9912
313/313 - 2s - loss: 0.0256 - accuracy: 0.9921 - 2s/epoch - 7ms/step
Test loss: 0.025640364736318588
Test accuracy: 0.9921000003814697

...

313/313 - 3s - loss: 2.3082 - accuracy: 0.0885 - 3s/epoch - 8ms/step
Untrained model, accuracy: 8.85%
Model: "sequential_6"

```

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_8 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_9 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_9 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_6 (Flatten)	(None, 1600)	0
dropout_4 (Dropout)	(None, 1600)	0
dense_8 (Dense)	(None, 10)	16010

```

=====
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0

313/313 - 2s - loss: 0.0235 - accuracy: 0.9933 - 2s/epoch - 7ms/step
Restored model, accuracy: 99.33%

```

Listing 2. Snippet of the output associated with Listing 1.

6. How are model states saved and loaded in PyTorch?

PyTorch supports saving and loading dictionaries called `state_dict` objects for models and optimizers in addition to saving and loading information such as epoch number, and training loss. It provides **`torch.save`** function that serializes and saves data related to a model state. An example of using this function for saving the state of a model is shown below:

```

torch.save({
    'epoch': EPOCH,
    'model_state_dict': cnn.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': LOSS,
}, PATH)

```

The saved model state can be loaded using the **`torch.load`** function and an example of the code snippet showing the use of this function is provided below:

```

restart = torch.load(PATH)
model.load_state_dict(restart['model_state_dict'])
optimizer.load_state_dict(restart['optimizer_state_dict'])
epoch = restart['epoch']
loss = restart['loss']

```

As per the PyTorch documentation the mechanism discussed in this section falls under the category of “general checkpointing”. The complete code showing the process of saving and loading the state of a CNN model when training with MNIST data is shown in Listing 3 and the associated output is shown in Listing 4.

```

import torch
from torchvision import datasets
from torchvision.transforms import ToTensor
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.autograd import Variable
import torch.nn as nn

traindata = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
testdata = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)

trainloader = torch.utils.data.DataLoader(traindata, batch_size=100, shuffle=True,
num_workers=2)
testloader = torch.utils.data.DataLoader(testdata, batch_size=100, shuffle=False,
num_workers=2)

class cnn(nn.Module):
    def __init__(self):
        super(cnn, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,

```

```

        padding=2,
    ),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
)
self.conv2 = nn.Sequential(
    nn.Conv2d(16, 32, 5, 1, 2),
    nn.ReLU(),
    nn.MaxPool2d(2),
)
self.out = nn.Linear(32 * 7 * 7, 10)
def forward(self, a):
    a = self.conv(a)
    a = self.conv2(a)
    a = a.view(a.size(0), -1)
    outp = self.out(a)
    return outp, a

model = cnn()
lossFct = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = 0.001)
PATH = "chkptnew.out"
num_epochs = 1
loss = 0.2

def train(num_epochs, model, loaders):
    model.train()
    # Train the model
    total_step = len(trainloader)
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(trainloader):
            b_x = Variable(images)    # batch x
            b_y = Variable(labels)    # batch y
            output = model(b_x)[0]
            loss = lossFct(output, b_y)
            # clear gradients for this training step
            optimizer.zero_grad()
            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()
            if (i+1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                      .format(epoch + 1, num_epochs, i + 1, total_step,
loss.item()))

```



```

        pass
    pass
pass
train(num_epochs, model, trainloader)
torch.save({
    'epoch': num_epochs,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}, PATH)

model2 = cnn()
optimizer = optim.Adam(model2.parameters(), lr = 0.001)
checkpoint = torch.load(PATH)
model2.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
print('Parameters loaded in model2')
print(model2)
print(epoch)
print(loss)

#continue working with model2 as needed
def test():
    # Test the model
    model2.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        i=0
        for images, labels in testloader:
            test_output, last_layer = model2(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
            if(i < 20) :
                plt.imshow(images[i].squeeze(), cmap='gray')
                #plt.title('%i' % labels[i])
                #plt.title ('%i' % pred_y[i])
                plt.show()
                print(f'Predicted number in the image : {pred_y[i]}')
                print(f'Actual number as per image label: {labels[i]}')
            i = i+1
    pass

```

```

        print('Test Accuracy of the model on the 10000 test images: %.2f' %
accuracy)

    pass

test()

```

Listing 3. Complete code demonstrating the process of saving and loading model states using PyTorch.

```

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 144131899.65it/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 3816739.99it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/raw/t10k-
images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 47284589.85it/s]
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw





Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/raw/t10k-
labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 770309.68it/s]
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
Epoch [1/1], Step [100/600], Loss: 0.2485
Epoch [1/1], Step [200/600], Loss: 0.2679
Epoch [1/1], Step [300/600], Loss: 0.1162
Epoch [1/1], Step [400/600], Loss: 0.0906
Epoch [1/1], Step [500/600], Loss: 0.0972
Epoch [1/1], Step [600/600], Loss: 0.0632
Parameters loaded in model2
cnn(
  (conv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
1
0.2

Predicted number in the image : 7
Actual number as per image label: 7

Predicted number in the image : 0
Actual number as per image label: 0

Predicted number in the image : 1
Actual number as per image label: 1

```

```

Predicted number in the image : 2
Actual number as per image label: 2

Predicted number in the image : 2
Actual number as per image label: 2

Predicted number in the image : 3
Actual number as per image label: 3
. . .

Predicted number in the image : 7
Actual number as per image label: 7
Test Accuracy of the model on the 10000 test images: 0.99
```

Listing 4. Output from the code shown in Listing 3.

7. How is checkpointing done in PyTorch?

PyTorch has beta version of packages for checkpointing models using both sequential and distributed computing paradigms and provides the following packages: `torch.utils.checkpoint` [6] and `torch.distributed.checkpoint` [7]. These packages and associated functions will be tested and information on them will be shared in future.

8. Summary:

Checkpointing is the process of saving the state of a model at a specific frequency and can be useful for recovering from interruptions during the model training process and in saving metrics from intermediate epochs. When training AI models with large-scale data sets on shared computing platforms, it is recommended to write checkpoints at optimal frequencies (i.e., after a certain number of epochs or after processing a certain number of samples) to recover from interruptions with minimum loss in time and computing cost while also accounting for the checkpointing overheads. Developers would need to consider the trade-offs between adopting different formats for saving the model states or using checkpointing depending upon the selected algorithms.

References:

1. Tensorflow, checkpoint: <https://www.tensorflow.org/guide/checkpoint>
2. Tensorflow, saved model: https://www.tensorflow.org/guide/saved_model
3. Tensorflow, save and load model:
https://www.tensorflow.org/tutorials/keras/save_and_load#:~:text=An%20entire%20model%20can%20be,be%20saved%20in%20HDF5%20format
4. Keras, save and load model: https://www.tensorflow.org/tutorials/keras/save_and_load
5. Keras, serialization and saving: https://www.tensorflow.org/guide/keras/serialization_and_saving
6. Pytorch Checkpoint: <https://pytorch.org/docs/stable/checkpoint.html>
7. PyTorch Distributed Checkpoints: https://pytorch.org/docs/stable/distributed_checkpoint.html
8. Keras examples: https://keras.io/examples/vision/mnist_convnet/
9. Nutan Sharma, PyTorch CNN with MNIST: <https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118>
10. PyTorch, saving and loading general checkpoints:
https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html