Lightweight Machine Learning Framework Performance

Kyle Hoffpauir, Adam Kolides

November 2022

Abstract

The Internet of Things is constantly expanding with new applications. One of these new applications is using lightweight machine learning models on them. In an effort to evaluate which lightweight machine learning model was the best to use with IoT devices, we looked at two models, Tensorflow Lite and MXNet. Models were evaluated on accuracy, cpu utilization, and runtime. Neural networks were created and trained using the MNIST dataset of handwritten digits. MXNext and Tensorflow neural networks were then converted into their lightweight counterparts Tensorflow lite and ONNX (respectively). The lightweight models were then run against a test set of MNIST data to evaluate their runtime, cpu utilization, and accuracy. A handmade novel dataset was created of 28x28 pixel digits and evaluated on accuracy in order to see the accuracy on a small novel dataset.

1 Introduction

The Internet of Things is an expanding market rooted in embedded systems. Applications are leveraging the new power of microcomputers combined with

machine learning to perform what is being called Edge Computing [1]. By placing sensors on the edge, data can be collected and sent to the cloud for processing and powerful results from a small internet connected device. This power can be further harnessed through an emerging paradigm of Edge Intelligence [2], placing the computing power of machine learning on these small IoT devices can create systems that are much more robust and advanced than traditional applications. However, this edge intelligence paradigm requires advancements of existing technology in the form of lightweight, portable systems. The market has thus grown with a new type of Artificial Intelligence system in the form of lightweight machine learning frameworks. These frameworks exist on the edge as sensors, using pre-trained models to classify incoming data in place before sending it to the cloud. This increases the overall power of the network and eases the strain on the cloud services hosting the edge intelligence model. We wanted to focus on three lightweight machine learning frameworks: Tensorflow Lite, Lasagne, and MXNet. We aimed to train these applications on the same dataset, a collection of pictures of handwritten digits and evaluate them based on how well they correctly predicted what the digit was. We used the Modified National Institute of Standards and Technology (MNIST) dataset for our project. We trained the applications to create a neural network and we evaluated them based on their cpu usage, accuracy, and speed. We determined that the ideal lightweight ML framework would produce a model which could accurately predict the correct handwritten digit using the least amount of computational force the fastest.

2 Related Works

Significant research efforts have been targeted at lightweight machine learning techniques and the Internet of Things (IoT). Deng et. al introduce the concept

of Edge Intelligence and outline the paradigm for how cloud computing technologies can be merged with edge devices in order to create a more powerful network[2]. The heterogeneity of the devices on an Edge Intelligence enabled platform creates many issues, leading to a need for a manner in which these different devices can communicate with one another. Rahman et. al create an ontology to simplify the communication between IoT devices and the cloud to push the Edge to Intelligence. [3] This sentiment is further explored in Bzai et. al's exploration of how machine learning can be implemented into IoT devices to have learning done on the edge, implementing the Edge Intelligence paradigm and evaluating the value of this from an industry prospective [1]. Further papers from Sajeev et. al[4] and Wang et. al[5] explore applications of Edge Intelligence in systems and how they can be built effectively. The following work aims to extend the foundational principle of edge intelligence with a simple implementation in order to further our own understanding of the systems and see how it could be implemented in a simple test environment.

3 Materials and Methods

In order to evaluate the performance of all of the lightweight ML frameworks, we first created python applications for each framework to do machine learning on the dataset. We had originally were planning on doing three models (Tensorflow Lite, MXNet, and Lasagne), but ended with two (Tensorflow Lite, and MXNet) because the Lasagne framework was not creating the framework the way we wanted it to. A significant portion of research time went to attempting to run the python files on a raspberry pi VM with limited memory and cpu resources, but getting the environment set up proved to take too much time and presented more issues than were reasonably able to be dealt with on the time table of this project. The neural networks were instead run on full size

machines. Tensorflow was chosen due to it's usability in terms of neural networks and the ability to convert to tensorflow lite. The tensorflow library is well equipped with functions that make it easy to spin up a neural network with only a few lines of code. The twensorflow model was then trained on the MNIST dataset and we used a test set to validate that training. The tensorflow model was then converted to tensorflow lite using the built-in TFLiteConverter. This conversion was easily done with the pre-built functions and we were able to save that pre-trained model so it would not have to be trained on the edge device. The TF lite model was then sent through the TFLite optmizer. This lowers the size of the floats in the model vectors from 32 to 16 along with optimizing the network for efficient space utilization. The model was then saved to be used later. The MXNet model was created in a similar way, making a neural network and training it on the MNIST dataset. Once the model was trained and validated using the training and testing sets provided from the MNIST dataset, we converted it to the ONNX lightweight frameowork. The ONNX framework is similar to the tensorflow lite lightweight ML framework, but leaves much to be desired in terms of documentation and useful functions. Tensorflow lite was well documented and easy to work with whereas ONNX was fairly archiac and required much more effort to get working. Once the lightweight ONNX model was createsd, we could then run our MNIST set on the models to measure their effectiveness. We were planning on doing a variety of machine learning techniques with each framework such as neural netowrks, classifications, decision trees, etc. but we ended up only creating a neural network for each, as that was the one ML technique that could be consistently created among the frameworks we chose to use. It seems that lightweight ML frameworks tend towards being used in neural networks as it is one of the more computationally intensive and largely implemented ML models.

Python code for Tensorflow framework:

```python
import numpy as np
import tensorflow as tf

#https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/lite/ex
# amples/experimental_new_converter/Keras_LSTM_fusion_Codelab.ipynb#scrollTo=-yEAraXGDlcQ

model = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(28, 28), name='input'),
    tf.keras.layers.LSTM(20, time_major=False, return_sequences=True),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax, name='output')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
print(model.summary())

# Load MNIST dataset.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.astype(np.float32)
x_test = x_test.astype(np.float32)

_EPOCHS = 5
if _FAST_TRAINING:
    _EPOCHS = 1
    _TRAINING_DATA_COUNT = 1000
    x_train = x_train[:_TRAINING_DATA_COUNT]
    y_train = y_train[:_TRAINING_DATA_COUNT]

model.fit(x_train, y_train, epochs=_EPOCHS)
model.evaluate(x_test, y_test, verbose=0)

run_model = tf.function(lambda x: model(x))
BATCH_SIZE = 1
STEPS = 28
INPUT_SIZE = 28
concrete_func = run_model.get_concrete_function(
    tf.TensorSpec([BATCH_SIZE, STEPS, INPUT_SIZE], model.inputs[0].dtype))
```

Python code to convert to Tensorflow Lite:

```python
MODEL_DIR = "keras_lstm"
model.save(MODEL_DIR, save_format="tf", signatures=concrete_func)

converter = tf.lite.TFLiteConverter.from_saved_model(MODEL_DIR)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)

# Run the model with TensorFlow to get expected results.
TEST_CASES = 10

# Run the model with TensorFlow Lite
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

for i in range(TEST_CASES):
    expected = model.predict(x_test[i:i+1])
    interpreter.set_tensor(input_details[0]["index"], x_test[i:i+1, :, :])
    interpreter.invoke()
    result = interpreter.get_tensor(output_details[0]["index"])

    # Assert if the result of TFLite model is consistent with the TF model.
    np.testing.assert_almost_equal(expected, result, decimal=5)
    print("Done. The result of TensorFlow matches the result of TensorFlow Lite.")

    interpreter.reset_all_variables()

converter.optimizations = [tf.lite.Optimize.DEFAULT]
#reduce the size of a floating point model by quantizing the weights to float16
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
#save the quanitized model toa binary file
open("model_quant_tl.tflite", "wb").write(tflite_quant_model)
```

```
import math
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import subprocess
from tensorflow.lite.python.lite import TFLiteConverter

def show_sample(images, labels, sample_count=25):
    # Create a square with can fit {sample_count} images
    grid_count = math.ceil(math.ceil(math.sqrt(sample_count)))
    grid_count = min(grid_count, len(images), len(labels))
    plt.figure(figsize=(2*grid_count, 2*grid_count))
    for i in range(sample_count):
        plt.subplot(grid_count, grid_count, i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(images[i], cmap=plt.cm.gray)
        plt.xlabel(labels[i])
    plt.show()

def run(dataset2):
    results = []
    for image, label in dataset2:
        # Load TFLite model and allocate tensors.
        interpreter = tf.lite.Interpreter(model_path="model_quant_tl.tflite")
        #allocate the tensors
        interpreter.allocate_tensors()
        interpreter.set_tensor(interpreter.get_input_details()[0]["index"], image)
        interpreter.invoke()
        output = interpreter.tensor(interpreter.get_output_details()[0]["index"])()[0]
        # Print the model's classification result
        digit = np.argmax(output)
        #print('Predicted Digit: %d\nConfidence: %f' % (digit, output[digit]))
        results.append(output==label)
    return len(results) / len(dataset2)
```

Python code for MXNet framework:

```
from __future__ import print_function
import mxnet as mx
from mxnet import gluon
from mxnet.gluon import nn
from mxnet import autograd as ag
from mxnet.contrib import onnx as onnx_mxnet
import numpy as np
import logging

mnist = mx.test_utils.get_mnist()

batch_size = 100
train_data = mx.io.NDArrayIter(mnist['train_data'], mnist['train_label'], batch_size, shuffle=True)
val_data = mx.io.NDArrayIter(mnist['test_data'], mnist['test_label'], batch_size)

net = nn.HybridSequential()
with net.name_scope():
    net.add(nn.Dense(128, activation='relu'))
    net.add(nn.Dense(64, activation='relu'))
    net.add(nn.Dense(10))

net.hybridize()

gpus = mx.test_utils.list_gpus()
ctx =  [mx.gpu()] if gpus else [mx.cpu(0), mx.cpu(1)]
net.initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.02})

epoch = 10
metric = mx.metric.Accuracy()
softmax_cross_entropy_loss = gluon.loss.SoftmaxCrossEntropyLoss()
for i in range(epoch):
    train_data.reset()
    for batch in train_data:
        data = gluon.utils.split_and_load(batch.data[0], ctx_list=ctx, batch_axis=0)
        label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx, batch_axis=0)
        outputs = []
        with ag.record():
            for x, y in zip(data, label):
```

```
                z = net(x)
                loss = softmax_cross_entropy_loss(z, y)
                loss.backward()
                outputs.append(z)
        metric.update(label, outputs)
        trainer.step(batch.data[0].shape[0])
    name, acc = metric.get()
    metric.reset()
    print('training acc at epoch %d: %s=%f'%(i, name, acc))

metric = mx.metric.Accuracy()
val_data.reset()
for batch in val_data:
    data = gluon.utils.split_and_load(batch.data[0], ctx_list=ctx, batch_axis=0)
    label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx, batch_axis=0)
    outputs = []
    for x in data:
        outputs.append(net(x))
    metric.update(label, outputs)
print('validation acc: %s=%f'%metric.get())
assert metric.get()[1] > 0.94

net.export("net", epoch=1)

converted_model_path = onnx_mxnet.export_model('./net-symbol.json', './net-0001.params', [(128,784)], np.double, 'converted.onnx')
```

Python code to convert to Onnx Model:

```
from __future__ import print_function
import argparse
import random
import sys
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR
import onnxruntime as rt
import math
import warnings
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import subprocess
import mxnet as mx
import onnx
from mxnet import gluon
from mxnet import autograd as ag
from mxnet.contrib import onnx as onnx_mxnet


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


def run(dataset2):
    results = []
    model = Net()
    state_dict = torch.load("mnist_cnn.pt")
    model.load_state_dict(state_dict)
    ##<All keys matched successfully>
    correct = 0
    #image, label = dataset2[random.randint(0, 9999)]
    for image, label in dataset2:
        # image = image.unsqueeze(0)
        image = image.unsqueeze(0)
        output = model(image)
        output = torch.argmax(output)
        #print(output, label, output == label)
        results.append(output==label)
    return len(results) / len(dataset2)
```

Once these models were trained to the best of their capabilities, the lightweight ML frameworks were implemented in a separate python application that represented the frameworks existing on the edge of the network as an IoT device

would be. We then run these lightweight models with a new selection of MNIST data so we could collect our metrics. The edge systems only have access to the input data and the pre-trained model, which we could easily import into the python application from the saved lightweight model. Each model received an influx of new mock "sensor data", which were more images of handwritten digits for the model to predict on. These models parse the data and use the pre-trained models they have from the cloud system to classify the data and produce a prediction. As the mock edge devices were running their machine learning, we kept recorded metrics concerning cpu usage, runtime, and accuracy in order to get an accurate, numerical representation of the effectiveness of the different systems. We then ran these tests once and averaged the metrics we collected in order to provide a more accurate metric. For another test, we created a folder of 18 handwritten digit samples that we created ourselves and ran them on our models in order to see the accuracy on singleton and small sample size data, simulating the IoT device being used by a single user or household.

4 Results

Our results were:

Model	**Accuracy (large set)**	**Accuracy (small set)**	**Cpu Utilization**	**Time (ms)**
Tensorflow Lite	100%	55.55%	50%	18.46
MXNet	100%	66.66%	54%	27.7

When we ran our small sample size tests, we got that the Tensorflow Lite model had an accuracy of 55.55% and the MXNet model had an accuracy of 66.66%.

5 Conclusions

In conclusion, between the Tensorflow Lite and MXNet models, we determined that Tensorflow Lite was the better lightweight framework. While the numebrs may betray this conclusion, we come to it from a careful consideration of all the information at our disposal. Our goal was to find a model that had a high accuracy percentage and a fast run time while also not being exceedingly taxing on the cpu. The MXNet model clearly has a higher accuracy percentage, we also need to take cpu utilization and runtime into consideration. Our end goal is to be able to run these on an IoT device, which often have minimal amounts of memory and low-end CPUs compared to larger computers, so even though MXNet has a higher accuracy, the Tensorflow Lite model ran faster and was less taxing to the cpu. This could be a result of the optimization function that was applied to the tensorflow model. The optimization may have improved the performance of the lightweight neural network, landing it in the winning position. The ONNX model had no such optimization and suffered slower runtime and a higher cpu usage. For these reasons, the TensorFlow lite model seems to be the better choice for implementing machine learning models on edge devices.

6 Future Work

For future work in this area, we recommend retesting the models which a larger dataset in order to get a better overall picture of the results. Our testing dataset only had 18 images in it, so increasing the number of images would give a more precise accuracy measurement. Additionally, we suggest using more models to broaden the scope and see which model is the best. Due to the time frame for this project, we could only successfully implement two models, but being able

to compare more models would provide a more well-rounded perspective on which is the best. We could only draw our conclusion based on the two models that we had, but maybe there is a better model that has a higher accuracy than our Tensorflow Lite model has but it just as lightweight as it. Finally, we were planning to run our models on a virtual machine that simulated being a Raspberry Pi, but we ran out of time to successfully implement it, but it would be interesting to see how these models behaved in this simulated environment.

References

[1] Jamal Bazi et al. Machine learning-enabled internet of things (iot): Data, applications, and industry perspective. *Electronics 11.17 (2022): 2626*, 2022.

[2] Shuiguang Deng et al. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal 7.8 (2020): 7457-7469*, 2020.

[3] Hafizur Rahman and Md. Iftekhar Hussain. A light-weight dynamic ontology for internet of things using machine learning technique. *ICT Express*, 7(3):355–360, 2021.

[4] G.P. Sajeev and M.P. Sebastian. Building semi-intelligent web cache systems with lightweight machine learning techniques. *Computers Electrical Engineering*, 39(4):1174–1191, 2013.

[5] Cong Wang, George Papadimitriou, Mariam Kiran, Anirban Mandal, and Ewa Deelman. Identifying execution anomalies for data intensive workflows using lightweight ml techniques. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.