
TensorState

Release 0.2.1

Oct 28, 2020

Contents

1	Introduction	1
1.1	Table of Contents	1
	Python Module Index	23
	Index	25

TensorState is a toolbox designed to analyze the way neural networks process information to calculate metrics about the network architecture.

Both Tensorflow and PyTorch are supported, but complex networks may prove problematic for some of the network functions (such as automatically building an efficiency model). Let us know if you run into problems.

Please send comments, suggestions, and feedback to nicholas.j.schaub@gmail.com.

1.1 Table of Contents

1.1.1 Installation

Introduction

TensorState uses accelerated Cython code to capture neural layer state information. This can create some issues when trying to install on architectures do not include prepackaged wheels. Please read the appropriate section carefully to make sure installation of the package is successful.

Most dependencies should be installed when using `pip`, however some may not be installed. Please contact me if there are any installation errors so I can fix it.

Windows and Python ≥ 3.6

Precompiled wheels exist for Windows 10 and versions of Python from 3.6 to 3.8. No special dependencies are required.

```
pip install TensorState
```

Linux

For Linux, there are manylinux wheels that should support most versions of Linux (`pip install TensorState`). In some cases it may try to compile from source (e.g. Alpine linux). When compiling, it is necessary to install `numpy` and `Cython` prior to installation.

```
pip install numpy==1.19.2 Cython==3.0a1
pip install TensorState
```

Mac OSX

There are currently no OSX wheels, so compilation from source is necessary following the instructions in the Linux section.

Troubleshooting

Currently, there is no fallback for working on platforms that do not have a C++ compiler or are working on platforms other than x86 architectures such as ARM. If there is interest, please [contact me](#).

The `TensorState` package is a work in progress, but the following tutorials demonstrate the current useful functionality of the package.

1.1.2 List of Tutorials

A Simple Tensorflow Tutorial

Table of Contents

- *Introduction*
- *Build and Train LeNet-5*
 - *Get MNIST*
 - *Create a LeNet-5 Model*
 - *Train the LeNet-5 Model*
- *Use TensorState to Evaluate LeNet-5*
- *Complete Example*

Introduction

The core ideas behind this package were originally described in our paper, [Assessing Intelligence in Artificial Neural Networks](#).

This package simplifies the capture of neural layers states, and provides some utility functions to assist in analyzing the state space of neural layers.

In this tutorial, we are going to build a classic convolutional neural network, LeNet-5. Then we are going to use `TensorState` to evaluate the network architecture, getting the efficiency of each layer and calculating the artificial intelligence quotient.

Build and Train LeNet-5

Get MNIST

To train this model, we need to get the MNIST data set. Fortunately, it comes packaged with Keras in Tensorflow. The original data was 8-bit and a single channel, so we need to add a channel axis and we are going to normalize the image to have pixel values ranging from 0-1.

```
import os

# Set the log level to hide some basic warning/info generated by Tensorflow
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# Fix for cudnn error on RTX gpus
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'

import tensorflow.keras as keras

# Load the data
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the data
train_images = train_images/255
test_images = test_images/255

# Add a channel axis
train_images = train_images[..., tf.newaxis]
test_images = test_images[..., tf.newaxis]
```

Create a LeNet-5 Model

The general structure of our LeNet-5 model will roughly follow the structure of the original network described by [LeCun et al.](#) However, we are going to make a few modifications to make it more modern relative to the original architecture, such as addition of l2 weight regularization, exponential linear units, and batch normalization. So, we start off by setting the Tensorflow random seed (to make the results reproducible) and set up the parameters of the convolutional layers.

```
import tensorflow as tf

# Set the random seed for reproducibility
tf.random.set_seed(0)

# Set the convolutional layer settings
reg = keras.regularizers.l2(0.0005)
kwargs = {'activation': 'elu',
          'kernel_initializer': 'he_normal',
          'kernel_regularizer': reg,
          'bias_regularizer': reg}
```

Next, we create the layers of the network. LeNet-5 has 2 convolutional layers and 2 fully connected layers. We will use max pooling layers after each convolutional layer, and we will add a batch normalization layer to all by the last fully connected layer.

```
# Build the layers
input_layer = keras.layers.Input(shape=(28,28,1), name='input')

# Unit 1
conv_1 = keras.layers.Conv2D(20, 5, name='conv_1', **kwargs)(input_layer)
norm_1 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_1)
maxp_1 = keras.layers.MaxPool2D((2,2), name='maxp_1')(norm_1)

# Unit 2
conv_2 = keras.layers.Conv2D(50, 5, name='conv_2', **kwargs)(maxp_1)
norm_2 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_2)
maxp_2 = keras.layers.MaxPool2D((2,2), name='maxp_2')(norm_2)

# Fully Connected
conv_3 = keras.layers.Conv2D(100, 4, name='conv_3', **kwargs)(maxp_2)
norm_3 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_3)

# Prediction
flatten = keras.layers.Flatten(name='flatten')(norm_3)
pred = keras.layers.Dense(10, name='pred')(flatten)

# Create the Keras model
model = keras.Model(
    inputs=input_layer,
    outputs=pred
)
```

Train the LeNet-5 Model

Next we train the LeNet-5 model, and stopping as soon as the validation accuracy stops increasing.

```
# Compile for training
model.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,
    ↪nesterov=True),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,
    ↪name='loss'),
    metrics=['accuracy']
)

# Stop the model once the validation accuracy stops going down
earlystop_callback = tf.keras.callbacks.EarlyStopping(
    monitor='val_accuracy',
    mode='max',
    patience=5,
    restore_best_weights=True
)

# Train the model
model.fit(
    train_images, train_labels, epochs=200,
    validation_data=(test_images, test_labels),
    batch_size=200,
    callbacks=[earlystop_callback],
    verbose=1
)
```


Use TensorState to Evaluate LeNet-5

To calculate neural layer efficiency, we need to capture the various states each layer takes on as the network processes data. This functionality is built into the `StateCapture` class, which is a Tensorflow layer that can be inserted into the model to automate the capturing of information passing through the network. The `StateCapture` layer acts like a probe that can be placed anywhere in the network: it records the information without modifying it, and passes it on to subsequent layers.

While `StateCapture` layers can be placed manually, there is a convenience function that can take an existing neural network and return a new network with `StateCapture` layers inserted at the designated areas. For example, we can attach a `StateCapture` layer to all convolutional layers.

```
import TensorState as ts
efficiency_model = ts.build_efficiency_model(model, attach_to=['Conv2D'], method='after
↪')
```

In the above code, we feed the trained LeNet-5 model into the function, designate we want to attach `StateCapture` layers to all 2D convolutional layers, and we want to capture the states after the layer. We could also capture the inputs going into and out of the layer by using `method='both'`. For more information on the `build_efficiency_model` method and additional settings, please see the [TensorState reference](#).

Now that the `efficiency_model` has been created, the `StateCapture` layers will collect all states of the network as images are fed to the network. Thus, to generate all possible states the network contains for the test data, we only need to predict the classes for the test data. Then we can look at how many states were collected for each layer.

```
predictions = efficiency_model.predict(train_images, batch_size=200)

for layer in efficiency_model.efficiency_layers:
    print('Layer {} number of states: {}'.format(layer.name, layer.state_count))
```

Note how `efficiency_model` has the efficiency layers stored in the `efficiency_layers` attribute of the model. The output of the above code should look something like this:

```
Layer conv_1_states number of states: 5760000
Layer conv_2_states number of states: 640000
Layer conv_3_states number of states: 10000
```

Since there are 10,000 images in the training data set, it is expected that the fully connected layer (`conv_3_states`) has 10,000 states recorded, since exactly one state will be recorded per image. The other layers are convolutional, generating multiple states per image. The number of states can be checked by determining the number of locations the convolutional operator is applied per image then multiplying by 10,000. For example, in a 28x28 image with a 5x5 convolutional operation performed on it, the dimensions of the output would be 24x24. Thus, the number of states for all 10,000 images would be $24*24*10,000=5,760,000$ states, which is the number of states observed by `conv_1_states`.

Note: The `state_count` is the raw number of states observed, and there are likely states that occur multiple times.

Now that the states of each layer have been captured, let's analyze the state space using the efficiency metric originally described by [Schaub et al.](#) The efficiency metric calculates the entropy of the state space and divides by the number of neurons in the layer, giving an efficiency value in the range 0.00-1.00.

```
for layer in efficiency_model.efficiency_layers:
    layer_efficiency = layer.efficiency()
    print('Layer {} efficiency: {:.1f}%'.format(layer.name, 100*layer_efficiency))
```

Next, we can calculate the artificial intelligence quotient (aIQ). Since things like neural network efficiency and aIQ are metrics calculated over the entire network, the StateCapture layer does not have built-in methods to calculate these values.

```
beta = 2 # fudge factor giving a slight bias toward accuracy over efficiency

print()
print('Network metrics...')
print('Beta: {}'.format(beta))

network_efficiency = ts.network_efficiency(efficiency_model)
print('Network efficiency: {:.1f}%'.format(100*network_efficiency))

accuracy = np.sum(np.argmax(predictions,axis=1)==train_labels)/train_labels.size
print('Network accuracy: {:.1f}%'.format(100*accuracy))

aIQ = ts.aIQ(network_efficiency,accuracy,beta)
print('aIQ: {:.1f}%'.format(100*aIQ))
```

Complete Example

```
import os

# Set the log level to hide some basic warning/info generated by Tensorflow
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# Fix for cudnn error on RTX gpus
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'

import tensorflow as tf
import tensorflow.keras as keras
import TensorState as ts
import numpy as np
import time

""" Load MNIST and transform it """
# Load the data
mnist = keras.datasets.mnist
(train_images,train_labels), (test_images,test_labels) = mnist.load_data()

# Normalize the data
train_images = train_images/255
test_images = test_images/255

# Add a channel axis
train_images = train_images[..., tf.newaxis]
test_images = test_images[..., tf.newaxis]

""" Create a LeNet-5 model """
# Set the random seed for reproducibility
tf.random.set_seed(0)

# Set the convolutional layer settings
reg = keras.regularizers.l2(0.0005)
kwargs = {'activation': 'elu',
          'kernel_initializer': 'he_normal',
```

(continues on next page)

(continued from previous page)

```

        'kernel_regularizer': reg,
        'bias_regularizer': reg}

# Build the layers
input_layer = keras.layers.Input(shape=(28,28,1), name='input')

# Unit 1
conv_1 = keras.layers.Conv2D(20, 5, name='conv_1', **kwargs)(input_layer)
norm_1 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_1)
maxp_1 = keras.layers.MaxPool2D((2,2), name='maxp_1')(norm_1)

# Unit 2
conv_2 = keras.layers.Conv2D(50, 5, name='conv_2', **kwargs)(maxp_1)
norm_2 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_2)
maxp_2 = keras.layers.MaxPool2D((2,2), name='maxp_2')(norm_2)

# Fully Connected
conv_3 = keras.layers.Conv2D(100, 4, name='conv_3', **kwargs)(maxp_2)
norm_3 = keras.layers.BatchNormalization(epsilon=0.00001, momentum=0.9)(conv_3)

# Prediction
flatten = keras.layers.Flatten(name='flatten')(norm_3)
pred = keras.layers.Dense(10, name='pred')(flatten)

# Create the Keras model
model = keras.Model(
    inputs=input_layer,
    outputs=pred
)

print(model.summary())

""" Train the model """
# Compile for training
model.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,
↪ nesterov=True),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, name=
↪ 'loss'),
    metrics=['accuracy']
)

# Stop the model once the validation accuracy stops going down
earlystop_callback = tf.keras.callbacks.EarlyStopping(
    monitor='val_accuracy',
    mode='max',
    patience=5,
    restore_best_weights=True
)

# Train the model
model.fit(
    train_images, train_labels, epochs=200,
    validation_data=(test_images, test_labels),
    batch_size=200,
    callbacks=[earlystop_callback],
    verbose=1

```

(continues on next page)

(continued from previous page)

```

    )

    """ Evaluate model efficiency """
    # Attach StateCapture layers to the model
    efficiency_model = ts.build_efficiency_model(model, attach_to=['Conv2D'], method='after
    ↪')

    # Collect the states for each layer
    print()
    print('Running model predictions to capture states...')
    start = time.time()
    predictions = efficiency_model.predict(test_images, batch_size=200)
    print('Finished in {:.3f}s!'.format(time.time() - start))

    # Count the number of states in each layer
    print()
    print('Getting the number of states in each layer...')
    for layer in efficiency_model.efficiency_layers:
        print('Layer {} number of states: {}'.format(layer.name, layer.state_count))

    # Calculate each layers efficiency
    print()
    print('Evaluating efficiency of each layer...')
    for layer in efficiency_model.efficiency_layers:
        start = time.time()
        print('Layer {} efficiency: {:.1f}% ({:.3f}s)'.format(layer.name, 100*layer.
    ↪efficiency(), time.time() - start))

    # Calculate the aIQ
    beta = 2 # fudge factor giving a slight bias toward accuracy over efficiency

    print()
    print('Network metrics...')
    print('Beta: {}'.format(beta))

    network_efficiency = ts.network_efficiency(efficiency_model)
    print('Network efficiency: {:.1f}%'.format(100*network_efficiency))

    accuracy = np.sum(np.argmax(predictions, axis=1) == test_labels) / test_labels.size
    print('Network accuracy: {:.1f}%'.format(100*accuracy))

    aIQ = ts.aIQ(network_efficiency, accuracy, beta)
    print('aIQ: {:.1f}%'.format(100*aIQ))

```

A Simple PyTorch Tutorial

Table of Contents

- *Introduction*
- *Build and Train LeNet-5*
 - *Get MNIST*
 - *Create a LeNet-5 Model*

– *Train the LeNet-5 Model*

- *Use TensorState to Evaluate LeNet-5*
- *Complete Example*

Introduction

The core ideas behind this package were originally described in our paper, [Assessing Intelligence in Artificial Neural Networks](#).

This package simplifies the capture of neural layers states, and provides some utility functions to assist in analyzing the state space of neural layers.

In this tutorial, we are going to build a classic convolutional neural network, LeNet-5. Then we are going to use TensorState to evaluate the network architecture, getting the efficiency of each layer and calculating the artificial intelligence quotient.

Build and Train LeNet-5

Get MNIST

To train this model, we need to get the MNIST data set. The data comes already rescaled to floats ranging between 0-1, so no rescaling is required.

```
from pathlib import Path
import requests, pickle, gzip
import torch
from torch.utils.data import TensorDataset, DataLoader

# Set up the directories
DATA_PATH = Path("data")
PATH = DATA_PATH/"mnist"
PATH.mkdir(parents=True, exist_ok=True)

# Download the data if it doesn't exist
URL = "http://deeplearning.net/data/mnist/"
FILENAME = "mnist.pkl.gz"
if not (PATH / FILENAME).exists():
    content = requests.get(URL + FILENAME).content
    (PATH / FILENAME).open("wb").write(content)

# Load the data
with gzip.open((PATH / FILENAME).as_posix(), "rb") as f:
    ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f, encoding="latin-1")

    x_train, y_train, x_valid, y_valid = map(
        torch.tensor, (x_train, y_train, x_valid, y_valid)
    )

    train_ds = TensorDataset(x_train, y_train)
    train_dl = DataLoader(train_ds, batch_size=200, shuffle=True)
    valid_ds = TensorDataset(x_valid, y_valid)
    valid_dl = DataLoader(valid_ds, batch_size=200)
```

Create a LeNet-5 Model

The general structure of our LeNet-5 model will roughly follow the structure of the original network described by [LeCun et al.](#) However, we are going to make a few modifications to make it more modern relative to the original architecture, such as addition of l2 weight regularization, exponential linear units, and batch normalization. So, we start off by setting the Tensorflow random seed (to make the results reproducible). Then we build the LeNet-5 class to define out network.

LeNet-5 has 2 convolutional layers and 2 fully connected layers. We will use max pooling layers after each convolutional layer, and we will add a batch normalization layer to all by the last fully connected layer.

```
import torch.nn as nn

# Set the random seed for reproducibility
torch.manual_seed(0)

# Build the layers
class LeNet5(nn.Module):

    def __init__(self):
        super().__init__()

        # Unit 1
        self.conv_1 = nn.Conv2d(1, 20, kernel_size=5, stride=1)
        torch.nn.init.kaiming_normal_(self.conv_1.weight)
        torch.nn.init.zeros_(self.conv_1.bias)
        self.elu_1 = nn.ELU()
        self.norm_1 = nn.BatchNorm2d(20,eps=0.00001,momentum=0.9)
        self.maxp_1 = nn.MaxPool2d(2,stride=2)

        # Unit 2
        self.conv_2 = nn.Conv2d(20, 50, kernel_size=5, stride=1)
        torch.nn.init.kaiming_normal_(self.conv_2.weight)
        torch.nn.init.zeros_(self.conv_2.bias)
        self.elu_2 = nn.ELU()
        self.norm_2 = nn.BatchNorm2d(50,eps=0.00001,momentum=0.9)
        self.maxp_2= nn.MaxPool2d(2,stride=2)

        # Fully Connected
        self.conv_3 = nn.Conv2d(50, 100, kernel_size=4, stride=1)
        torch.nn.init.kaiming_normal_(self.conv_3.weight)
        torch.nn.init.zeros_(self.conv_3.bias)
        self.elu_3 = nn.ELU()
        self.norm_3 = nn.BatchNorm2d(100,eps=0.00001,momentum=0.9)

        # Prediction
        self.flatten = nn.Flatten()
        self.pred = nn.Linear(100,10)
        torch.nn.init.kaiming_normal_(self.pred.weight)
        torch.nn.init.zeros_(self.pred.bias)

    def forward(self,data):
        x = data.view(-1, 1, 28, 28)
        x = self.conv_1(x)
        x = self.maxp_1(self.norm_1(self.elu_1(x)))
        x = self.conv_2(x)
        x = self.maxp_2(self.norm_2(self.elu_2(x)))
```

(continues on next page)

(continued from previous page)

```

        x = self.conv_3(x)
        x = self.norm_3(self.elu_3(x))
        x = self.pred(self.flatten(x))
        return x.view(-1, x.size(1))

# Set the device to run the model on (gpu if available, cpu otherwise)
dev = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

# Create the Keras model
model = LeNet5().to(dev)

```

Train the LeNet-5 Model

First, set up the parameters used for training. This will be set up to run with early stopping, similar to how Tensorflow has an early stopping callback. The `patience` parameter determines how many epochs to let past after the highest accuracy value is observed.

```

import torch.optim as optim

num_epochs = 200
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9,
                       weight_decay=0.0005, nesterov=True)
last_valid_accuracy = 0
val_count = 0
patience = 5

```

Next, create the function to process training and evaluation for all samples.

```

def epoch_func(x, y, train=False):
    predictions = model(x)
    num = len(x)
    accuracy = (torch.argmax(predictions, axis=1) == y).float().sum() / num
    loss = loss_func(predictions, y)

    if train:
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    return loss, accuracy, num

```

Finally, run the training and evaluation loop.

```

import time

for epoch in range(num_epochs):
    start = time.time()
    model.train()
    losses, accuracies, nums = zip(
        *[epoch_func(xb.to(dev), yb.to(dev), True) for xb, yb in train_dl]
    )
    train_loss = np.sum(np.multiply(losses, nums)) / np.sum(nums)
    train_accuracy = np.sum(np.multiply(accuracies, nums)) / np.sum(nums)

```

(continues on next page)

(continued from previous page)

```

model.eval()
with torch.no_grad():
    losses, accuracies, nums = zip(
        *[epoch_func(xb.to(dev), yb.to(dev), False) for xb, yb in valid_dl]
    )
    valid_loss = np.sum(np.multiply(losses, nums)) / np.sum(nums)
    valid_accuracy = np.sum(np.multiply(accuracies, nums)) / np.sum(nums)

    print('Epoch {}/{} ( {:.2f}s): TrainLoss={:.4f}, TrainAccuracy={:.2f}%, ValidLoss=
    ↪ {:.4f}, ValidAccuracy={:.2f}%'.format(
        str(epoch+1).zfill(3), num_epochs, time.time()-start,
        train_loss, 100*train_accuracy, valid_loss, 100*valid_accuracy
    ))

    # Early stopping criteria
    if valid_accuracy > last_valid_accuracy:
        val_count = 0
        last_valid_accuracy = valid_accuracy
    else:
        val_count += 1

    if val_count >= patience:
        break

```

Use TensorState to Evaluate LeNet-5

To calculate neural layer efficiency, we need to capture the various states each layer takes on as the network processes data. This functionality is built into the `StateCaptureHook` class, which is a hook that can be called before or after the designated layers to automate the capturing of information passing through the network. The `StateCaptureHook` acts like a probe that can be placed anywhere in the network: it records the information without modifying it, and passes it on to subsequent layers.

While `StateCaptureHook`'s can be placed manually, there is a convenience function that automatically adds hooks at the designated layers. For example, we can attach a `StateCaptureHook` to all convolutional layers.

```

import TensorState as ts
efficiency_model = ts.build_efficiency_model(model, attach_to=['Conv2d'], method='after
    ↪ ')

```

In the above code, we feed the trained LeNet-5 model into the function, designate we want to attach `StateCaptureHook`'s to all 2D convolutional layers, and we want to capture the states after the layer. We could also capture the inputs going into and out of the layer by using `method='both'`. For more information on the `build_efficiency_model` method and additional settings, please see the TensorState reference.

Now that the `efficiency_model` has been created, the `StateCaptureHooks` will collect all states of the network as images are fed to the network. Thus, to generate all possible states the network contains for the test data, we only need to evaluate the test data. Then we can look at how many states were collected for each layer.

```

model.eval()
with torch.no_grad():
    losses, accuracies, nums = zip(
        *[epoch_func(xb.to(dev), yb.to(dev), False) for xb, yb in valid_dl]
    )

```

(continues on next page)

(continued from previous page)

```
for layer in efficiency_model.efficiency_layers:
    print('Layer {} number of states: {}'.format(layer.name, layer.state_count))
```

Note how `efficiency_model` has the efficiency layers stored in the `efficiency_layers` attribute of the model. The output of the above code should look something like this:

```
Layer conv_1_post_states number of states: 5760000
Layer conv_2_post_states number of states: 640000
Layer conv_3_post_states number of states: 10000
```

Since there are 10,000 images in the training data set, it is expected that the fully connected layer (`conv_3_post_states`) has 10,000 states recorded, since exactly one state will be recorded per image. The other layers are convolutional, generating multiple states per image. The number of states can be checked by determining the number of locations the convolutional operator is applied per image then multiplying by 10,000. For example, in a 28x28 image with a 5x5 convolutional operation performed on it, the dimensions of the output would be 24x24. Thus, the number of states for all 10,000 images would be $24*24*10,000=5,760,000$ states, which is the number of states observed by `conv_1_post_states`.

Note: The `state_count` is the raw number of states observed, and there are likely states that occur multiple times.

Now that the states of each layer have been captured, let's analyze the state space using the efficiency metric originally described by [Schaub et al.](#) The efficiency metric calculates the entropy of the state space and divides by the number of neurons in the layer, giving an efficiency value in the range 0.00-1.00.

```
for layer in efficiency_model.efficiency_layers:
    layer_efficiency = layer.efficiency()
    print('Layer {} efficiency: {:.1f}%'.format(layer.name, 100*layer_efficiency))
```

Next, we can calculate the artificial intelligence quotient (aIQ). Since things like neural network efficiency and aIQ are metrics calculated over the entire network, the `StateCaptureHook`'s do not have built-in methods to calculate these values.

```
beta = 2 # fudge factor giving a slight bias toward accuracy over efficiency

print()
print('Network metrics...')
print('Beta: {}'.format(beta))

network_efficiency = ts.network_efficiency(efficiency_model)
print('Network efficiency: {:.1f}%'.format(100*network_efficiency))

accuracy = np.sum(np.multiply(accuracies, nums)) / np.sum(nums)
print('Network accuracy: {:.1f}%'.format(100*accuracy))

aIQ = ts.aIQ(network_efficiency, accuracy.cpu().item(), beta)
print('aIQ: {:.1f}%'.format(100*aIQ))
```

Complete Example

```
import requests, pickle, gzip, time

from pathlib import Path
```

(continues on next page)

(continued from previous page)

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import numpy as np

import TensorState as ts

# Set the device to run the model on (gpu if available, cpu otherwise)
dev = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

""" Load MNIST and transform it """
# Set up the directories
DATA_PATH = Path("data")
PATH = DATA_PATH/"mnist"
PATH.mkdir(parents=True, exist_ok=True)

# Download the data if it doesn't exist
URL = "http://deeplearning.net/data/mnist/"
FILENAME = "mnist.pkl.gz"
if not (PATH / FILENAME).exists():
    content = requests.get(URL + FILENAME).content
    (PATH / FILENAME).open("wb").write(content)

# Load the data
with gzip.open((PATH / FILENAME).as_posix(), "rb") as f:
    ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f, encoding="latin-1")

    x_train, y_train, x_valid, y_valid = map(
        torch.tensor, (x_train, y_train, x_valid, y_valid)
    )

    train_ds = TensorDataset(x_train, y_train)
    train_dl = DataLoader(train_ds, batch_size=200, shuffle=True)
    valid_ds = TensorDataset(x_valid, y_valid)
    valid_dl = DataLoader(valid_ds, batch_size=200)

""" Create a LeNet-5 model """
# Set the random seed for reproducibility
torch.manual_seed(0)

# Build the layers
class LeNet5(nn.Module):

    def __init__(self):
        super().__init__()

        # Unit 1
        self.conv_1 = nn.Conv2d(1, 20, kernel_size=5, stride=1)
        torch.nn.init.kaiming_normal_(self.conv_1.weight)
        torch.nn.init.zeros_(self.conv_1.bias)
        self.elu_1 = nn.ELU()
        self.norm_1 = nn.BatchNorm2d(20, eps=0.00001, momentum=0.9)
        self.maxp_1 = nn.MaxPool2d(2, stride=2)

        # Unit 2
        self.conv_2 = nn.Conv2d(20, 50, kernel_size=5, stride=1)

```

(continues on next page)

(continued from previous page)

```

torch.nn.init.kaiming_normal_(self.conv_2.weight)
torch.nn.init.zeros_(self.conv_2.bias)
self.elu_2 = nn.ELU()
self.norm_2 = nn.BatchNorm2d(50,eps=0.00001,momentum=0.9)
self.maxp_2= nn.MaxPool2d(2, stride=2)

# Fully Connected
self.conv_3 = nn.Conv2d(50, 100, kernel_size=4, stride=1)
torch.nn.init.kaiming_normal_(self.conv_3.weight)
torch.nn.init.zeros_(self.conv_3.bias)
self.elu_3 = nn.ELU()
self.norm_3 = nn.BatchNorm2d(100,eps=0.00001,momentum=0.9)

# Prediction
self.flatten = nn.Flatten()
self.pred = nn.Linear(100,10)
torch.nn.init.kaiming_normal_(self.pred.weight)
torch.nn.init.zeros_(self.pred.bias)

def forward(self, data):
    x = data.view(-1, 1, 28, 28)
    x = self.conv_1(x)
    x = self.maxp_1(self.norm_1(self.elu_1(x)))
    x = self.conv_2(x)
    x = self.maxp_2(self.norm_2(self.elu_2(x)))
    x = self.conv_3(x)
    x = self.norm_3(self.elu_3(x))
    x = self.pred(self.flatten(x))
    return x.view(-1, x.size(1))

# Create the Keras model
model = LeNet5().to(dev)

""" Train the model """
num_epochs = 200
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9,
                       weight_decay=0.0005, nesterov=True)
last_valid_accuracy = 0
val_count = 0
patience = 5

def epoch_func(x,y,train=False):
    predictions = model(x)
    num = len(x)
    accuracy = (torch.argmax(predictions,axis=1)==y).float().sum()/num
    loss = loss_func(predictions,y)

    if train:
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    return loss, accuracy, num

for epoch in range(num_epochs):
    start = time.time()

```

(continues on next page)

(continued from previous page)

```

model.train()
losses, accuracies, nums = zip(
    *[epoch_func(xb.to(dev), yb.to(dev), True) for xb, yb in train_dl]
)
train_loss = np.sum(np.multiply(losses,nums))/np.sum(nums)
train_accuracy = np.sum(np.multiply(accuracies,nums))/np.sum(nums)

model.eval()
with torch.no_grad():
    losses, accuracies, nums = zip(
        *[epoch_func(xb.to(dev), yb.to(dev), False) for xb, yb in valid_dl]
    )
valid_loss = np.sum(np.multiply(losses,nums))/np.sum(nums)
valid_accuracy = np.sum(np.multiply(accuracies,nums))/np.sum(nums)

print('Epoch {}/{ } ({:.2f}s): TrainLoss={:.4f}, TrainAccuracy={:.2f}%, ValidLoss=
→ {:.4f}, ValidAccuracy={:.2f}%'.format(
    str(epoch+1).zfill(3), num_epochs, time.time()-start,
    train_loss, 100*train_accuracy, valid_loss, 100*valid_accuracy
))

# Early stopping criteria
if valid_accuracy > last_valid_accuracy:
    val_count = 0
    last_valid_accuracy = valid_accuracy
else:
    val_count += 1

if val_count >= patience:
    break

""" Evaluate model efficiency """
# Attach StateCapture layers to the model
efficiency_model = ts.build_efficiency_model(model, attach_to=['Conv2d'], method='after
→ ')

# Collect the states for each layer
print()
print('Running model predictions to capture states...')
start = time.time()
model.eval()
with torch.no_grad():
    losses, accuracies, nums = zip(
        *[epoch_func(xb.to(dev), yb.to(dev), False) for xb, yb in valid_dl]
    )
print('Finished in {:.3f}s!'.format(time.time() - start))

# Count the number of states in each layer
print()
print('Getting the number of states in each layer...')
for layer in efficiency_model.efficiency_layers:
    print('Layer {} number of states: {}'.format(layer.name, layer.state_count))

# Calculate each layers efficiency
print()
print('Evaluating efficiency of each layer...')
for layer in efficiency_model.efficiency_layers:

```

(continues on next page)

(continued from previous page)

```

    start = time.time()
    print('Layer {} efficiency: {:.1f}% ({:.3f}s)'.format(layer.name, 100*layer.
    ↪ efficiency(), time.time() - start))

# Calculate the aIQ
beta = 2 # fudge factor giving a slight bias toward accuracy over efficiency

print()
print('Network metrics...')
print('Beta: {}'.format(beta))

network_efficiency = ts.network_efficiency(accuracy_model)
print('Network efficiency: {:.1f}%'.format(100*network_efficiency))

accuracy = np.sum(np.multiply(accuracies, nums)) / np.sum(nums)
print('Network accuracy: {:.1f}%'.format(100*accuracy))

aIQ = ts.aIQ(network_efficiency, accuracy.cpu().item(), beta)
print('aIQ: {:.1f}%'.format(100*aIQ))

```

1.1.3 Reference

tensorstate.TensorState

The `TensorState` module contains many of the core, high level functions that have been developed through state space research.

aIQ (*net_efficiency, accuracy, weight*)

Calculate the artificial intelligence quotient

The artificial intelligence quotient (aIQ) is a simple metric to report a balance of neural network efficiency and task performance. Although not required, it is assumed that the accuracy argument is a float ranging from 0.0-1.0, with 1.0 meaning more accurate.

$$aIQ = (net_efficiency * accuracy ** weight) ** (1/(weight+1))$$

The weight argument is an integer, with higher values giving more weight to the accuracy of the model.

Parameters

- **net_efficiency** (*[float]*) – A float ranging from 0.0-1.0
- **accuracy** (*[float]*) – A float ranging from 0.0-1.0
- **weight** (*[int]*) – An integer with value ≥ 1

Raises `ValueError` – Raised if $weight \leq 0$

Returns The artificial intelligence quotient

Return type `[float]`

build_efficiency_model (*model, attach_to, exclude=[], method='after', storage_path=None*)

Attach state capture methods to a neural network

This method takes an existing neural network model and attaches either layers or hooks to the model to capture the states of neural network layers.

For Tensorflow, only `keras.Model` networks can serve as inputs to this function. When a Tensorflow model is fed into this function, a new network is returned where `StateCapture` layers are inserted into the network at the designated locations.

For PyTorch, a neural network that implements the `Module` class will have hooks added to the layers. A new network is not generated, but for consistency the model is returned from this function.

Parameters

- **model** (`[keras.Model, torch.nn.Module]`) – A Keras model
- **attach_to** (`(list, optional)`) – List of strings indicating the types of layers to attach to. Names of layers can also be specified to attach `StateCapture` to specific layers
- **exclude** (`(list, optional)`) – List of strings indicating the names of layers to not attach `StateCapture` layers to. This will override the `attach_to` keyword, so that a `Conv2D` layer with the name specified by `exclude` will not have a `StateCapture` layer attached to it. Defaults to `[]`.
- **method** (`(str, optional)`) – The location to attach the `StateCapture` layer to. Must be one of `['before', 'after', 'both']`. Defaults to `'after'`.
- **storage_path** (`([str, pathlib.Path], optional)`) – Path on disk to store states in `zarr` format. If `None`, states are stored in memory. Defaults to `None`.

Returns A model of the same type as the input model

Return type `model`

entropy (`counts, alpha=1`)

Calculate the Renyi entropy

The Renyi entropy is a general definition of entropy that encompasses Shannon's entropy, Hartley (maximum) entropy, and min-entropy. It is defined as:

$$(1-\alpha)^{-1} * \log_2(\sum(p^{\alpha}))$$

By default, this method sets `alpha=1`, which is Shannon's entropy.

Parameters

- **counts** (`numpy.ndarray`) – Array of counts representing number of times a state is observed.
- **alpha** (`([int, float], optional)`) – Entropy order. Defaults to `1`.

Returns The entropy of the count data.

Return type `[float]`

network_efficiency (`efficiencies`)

Calculate the network efficiency

This method calculates the neural network efficiency, defined as the geometric mean of the efficiency values calculated for the network.

Parameters **efficiencies** (`([list, keras.Model, torch.nn.Module])`) – A list of efficiency values (floats) or a `keras.Model`

Returns The network efficiency

Return type `[float]`

reset_efficiency_model (`model`)

Reset all efficiency layers/hooks in a model

This method resets all efficiency layers or hooks in a model, setting the `state_count=0`. This is useful for repeated evaluation of a model during a single session.

Parameters `model` (`[keras.Model, torch.nn.Module]`) – Model to reset

tensorstate.Layers

The `Layers` module contains classes that interact with neural network layers.

class AbstractStateCapture (`name, disk_path=None, **kwargs`)

Bases: `abc.ABC`

Base class for capturing state space information in a neural network.

This class implements the infrastructure used to capture, quantize, and process state space information. For Tensorflow, a subclass is constructed to inherit these methods as a layer to be inserted into the network. For PyTorch, a subclass is constructed to implement these methods as layer hooks.

This class captures state information and quantizes layer outputs as firing or not firing based on whether the values are >0 or ≤ 0 respectively. Although this layer is intended to be attached before or after a neural layer, this can actually be attached to any layer type. After recording the firing state of all neurons, the original input is returned unaltered. Thus, this layer can be thought of as a “probe”, since it does not add or subtract from the function of a network.

Layer states are stored in a zarr array, which permits compressed storage of data in memory or on disk. Only blosc compression is used to ensure fast compression/decompression speeds. By default, data is stored in memory, but data can be stored on disk to reduce memory consumption by using the `disk_path` keyword.

NOTE: This layer currently only works within Tensorflow Keras models and PyTorch models.

counts ()

Layer state counts

This method returns a `numpy.array` of integers, where each integer is the number of times a state is observed. The identity of the states can be obtained by calling the `state_ids` method.

NOTE: The list only contains counts for observed states, so all values will be >0

Returns Counts of stat occurrences

Return type `[list of int]`

efficiency (`alpha1=1, alpha2=None`)

Calculate the efficiency of the layer

This method returns the efficiency of the layer. Originally, the efficiency was defined as the ratio of Shannon’s entropy to the theoretical maximum entropy based on the number of neurons in the layer. This method with no inputs will return that value. However, this method will also now permit defining the `alpha` value for the Renyi entropy, so that the efficiency will be calculated as the Renyi entropy of order `alpha1` divided by the maximum theoretical entropy.

Parameters

- **alpha1** (`[float, int], optional`) – Order of Renyi entropy in numerator
- **alpha2** (`[float, int, None], optional`) – Order of Renyi entropy in denominator

Returns The efficiency of the layer

Return type `[float]`

entropy (*alpha=1*)

Calculate the entropy of the layer

Calculate the entropy from the observed states. The alpha value is the order of entropy calculated using the formula for Renyi entropy. When alpha=1, this returns Shannon's entropy.

Parameters **alpha** (*int, None*) – Order of entropy to calculate. If None, then use `max_entropy()`

Returns The entropy of the layer

Return type [float]

max_entropy ()

Theoretical maximum entropy for the layer

The maximum entropy for the layer is equal to the number of neurons in the layer. This is different than the maximum entropy value that would be returned from the `TensorState.entropy` method with `alpha=0`, which is a count of the observed states.

Returns Theoretical maximum entropy value

Return type [float]

raw_states

Raw state data as stored in memory, bit compressed

reset_states (*input_shape=None*)

Initialize the state space

This method initializes the layer and resets any previously held data. The zarr array is initialized in this method.

Parameters **input_shape** (*TensorShape, tuple, list*) – Shape of the input.

state_count

The total number of observed states, including repeats.

state_ids ()

Identity of observed states

This method returns a list of byte arrays. Each byte array corresponds to a unique observed state, where each bit in the byte array corresponds to a neuron. The list returned by this method matches the list returned by `counts`, so that the value in `state_ids` at position `i` is associated with the `counts` value at position `i`.

For example, if the StateCapture layer is attached to a convolutional layer with 8 neurons, then each item in the list will be a byte array of length 1. If one of the bytes is `\x00` (a null byte), then the state has no firing neurons.

NOTE: Only observed states are contained in the list.

Returns Unique states observed by the layer

Return type [list of Bytes]

states

Decompressed state data

class StateCapture (*name, disk_path=None, **kwargs*)Bases: `TensorState.Layers.AbstractStateCapture`

Tensorflow keras layer to capture states in keras models

This class is designed to be used in a Tensorflow keras model to automate the capturing of neurons states as data is passed through the network.

build (*input_shape*)

Build the StateCapture Keras Layer

This method initializes the layer and resets any previously held data. The zarr array is initialized in this method.

Parameters *input_shape* (*TensorShape*) – Either a *TensorShape* or list of *TensorShape* instances.

call (*inputs*)

class *StateCaptureHook* (*name*, *disk_path=None*, ***kwargs*)

Bases: *TensorState.Layers.AbstractStateCapture*

StateCapture hook for PyTorch

This class implements all methods in *AbstractStateCapture*, but is designed to be a pre or post hook for a layer.

tensorstate.States

The *States* module contains functions that operate on state space.

compress_states (*states*)

Compress a state space tensors

This function quantizes neurons into firing (>0) or non-firing (≤ 0), then compresses the bits into uint8 values. Thus, if a layer has 8 neurons in it, then the output is a raw byte ranging in value from 0-255. This compresses the statespace by 32x relative to holding values as floats, or by 8x relative to holding values as boolean. This is an important consideration since state space is large and grows exponentially with the number of neurons in the layer.

Parameters *states* (*[numpy.ndarray]*) – A 2d array of neuron outputs as *numpy.float32* or *np.bool* values, where columns are a particular neuron's value, and rows are states.

Returns

A 2d array of uint8 values, where each value is the compressed representation of the state.

Return type *numpy.ndarray*

decompress_states (*states*, *num_neurons*)

Decompress states to numpy array of booleans

This functions takes a 2d numpy array of compressed neuron states and returns a boolean array of states, where each column of values represents the state of an individual neuron (firing=True, non-firing=False).

For example, take a neuron layer with 5 neurons. The compressed state will be represented by a single byte, and if all but the first neuron is firing then the bits will be set as follows:

'00011110'

To decompress this, the number of neurons needs to be input to know how many of the bits are actual neuron representations. When this state is decompressed, the numpy array will be:

[False, True, True, True, True]

Parameters

- **states** (*[numpy.ndarray]*) – A 2d array of compressed states. See *compress_states* function.

- **num_neurons** (*[int]*) – The number of neurons in the layer.

Returns Boolean numpy array of neuron states

Return type decompressed_states ([np.ndarray])

sort_states (*states, state_count*)

Sort the states to place identical states next to each other

This function sorts the states stored in a 2d numpy.ndarray so that identical states are placed next to each other. To increase speed, the states are not actually sorted since moving data around in memory can be time consuming, and usually not useful. What is returned is a sorted index and the location of unique states in the sorted index.

Parameters

- **states** (*[numpy.ndarray]*) – A 2d array of compressed states. See `compress_states` function.
- **state_count** (*[int]*) – The number of states (or number of rows to sort).

Returns

Bin edges, or locations of unique states index ([np.ndarray]): Sorted index. This output can be used to actually

sort the input states by doing `states[index]`

Return type edges ([np.ndarray])

1.1.4 What is state space?

An Abstract Explanation

Traditionally, the fundamental unit of computation in the human nervous system has been the neuron. Some of the early thought in how the brain processes information suggested that particular neurons encode particular information, which may be called features in the data science community. Thus, if we have ~86 billion neurons, then we can roughly learn 86 billion features. However, as human intelligence progressed it was quickly discovered that even simple tasks often recruit multiple, seemingly unrelated areas of the brain. This led to the idea that the meaning of an individual neuron may not matter as much as how neurons fire as a collective, suggesting the number of neurons are an exponent rather than a coefficient to a human's capacity to learn new tasks.

Recent research into artificial neural networks and how they function approach the subject in the antiquated approach to understanding human neurons, where each neuron encodes a particular feature or function. The idea of state space diverges and challenges this conception of neurons, and attempts to show how neurons in artificial neural networks operate in parallel rather than discrete units in the same way that the human brain operates. In this conception of a neuron layer, the features are encoded in the state of firing neurons rather than individual neurons. Thus, in the current conception of neural networks, a layer with 16 neurons would encode 16 features, but in the state space up to 2^{16} features can be encoded (assuming neurons are either firing or not firing).

t

`TensorState.Layers`, [19](#)
`TensorState.States`, [21](#)
`TensorState.TensorState`, [17](#)

A

`AbstractStateCapture` (class in `TensorState.Layers`), 19

`aiQ()` (in module `TensorState.TensorState`), 17

B

`build()` (*StateCapture method*), 21

`build_efficiency_model()` (in module `TensorState.TensorState`), 17

C

`call()` (*StateCapture method*), 21

`compress_states()` (in module `TensorState.States`), 21

`counts()` (*AbstractStateCapture method*), 19

D

`decompress_states()` (in module `TensorState.States`), 21

E

`efficiency()` (*AbstractStateCapture method*), 19

`entropy()` (*AbstractStateCapture method*), 19

`entropy()` (in module `TensorState.TensorState`), 18

M

`max_entropy()` (*AbstractStateCapture method*), 20

N

`network_efficiency()` (in module `TensorState.TensorState`), 18

R

`raw_states` (*AbstractStateCapture attribute*), 20

`reset_efficiency_model()` (in module `TensorState.TensorState`), 18

`reset_states()` (*AbstractStateCapture method*), 20

S

`sort_states()` (in module `TensorState.States`), 22

`state_count` (*AbstractStateCapture attribute*), 20

`state_ids()` (*AbstractStateCapture method*), 20

`StateCapture` (class in `TensorState.Layers`), 20

`StateCaptureHook` (class in `TensorState.Layers`), 21

`states` (*AbstractStateCapture attribute*), 20

T

`TensorState.Layers` (module), 19

`TensorState.States` (module), 21

`TensorState.TensorState` (module), 17