Metal for Accelerating Machine Learning

Session 609

Anna Tikhonova, GPU Software Engineer
Metal Performance Shaders

GPU-accelerated primitives, optimized for iOS and macOS

• Image processing
• Linear algebra
• Machine learning—inference

Using Metal 2 for Compute  WWDC 2017

What’s New in Metal, Part 2  WWDC 2016
Metal Performance Shaders

GPU-accelerated primitives, optimized for iOS and macOS

• Image processing
• Linear algebra
• Machine learning—inference and training
Metal Performance Shaders

GPU-accelerated primitives, optimized for iOS and macOS

• Image processing
• Linear algebra
• Machine learning—inference and training
• Ray tracing
Training and Inference

Image classification example

Model

- cat
- rabbit
- dog
- giraffe
- ...
- ...
Training
Image classification example

CNN

Trained Parameters

Training

- cat
- rabbit
- dog
- giraffe

- horse
- dog
- rabbit
- giraffe
Training
Image classification example

Training

CNN

Trained Parameters

cat
rabbit
dog
giraffe

Training
Inference
Image classification example

CNN
Training

- cat
- rabbit
- dog
- giraffe

Trained Parameters
Inference
Image classification example

Input Image

CNN
Trained Parameters
- cat
- rabbit
- dog
- giraffe

CNN
Inference
- cat
CNN Inference Enhancements
FP16 accumulation

Available with Apple A11 Bionic GPU for
- Convolution
- Convolution transpose

Sufficient precision for commonly used neural networks
Delivers better performance than FP32
CNN Inference Enhancements
FP16 accumulation

```swift
let conv = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil),
                                 weights: MyWeights(withFile:"conv.dat"))
conv.accumulatorPrecisionOption = .half
```
CNN Inference Enhancements

FP16 accumulation

```swift
let conv = MPSConvolutionNode(source: MPSImageNode(handle: nil),
                               weights: MyWeights(withFile: "conv.dat"))

conv.accumulatorPrecisionOption = .half
```
CNN Training
Network
Handwritten digit recognition

*MNIST handwritten digit database, Yann LeCun, Corinna Cortes, Christopher J.C. Burges, 2013
Trained Parameters

For inference

Input 7

Convolution  ReLu  Max Pooling  Fully-Connected  Dropout  SoftMax

Output 7

Trained Parameters
Trained Parameters
For inference

Convolution and Fully-Connected Weights

Input 7

Convolution
ReLu
Max Pooling
Fully-Connected
Dropout
SoftMax

Output 7

Convolution and Fully-Connected Weights

Trained Parameters
Training
Initializing weights

Input

Training Parameters

Output
Training
Initializing weights

Input

Training Parameters

Output

-3.40e-02 -1.77e-01
-2.10e-03 -3.53e-02
-1.45e-01 -6.11e-02
-8.27e-02 -9.71e-02
-1.10e-01 -8.75e-02
-5.46e-02 -2.93e-02
-8.77e-03 -8.80e-02
-1.35e-01 -1.97e-01
-4.34e-02 -5.58e-02
-2.96e-02 -1.31e-01
-4.57e-02 -3.39e-02
-4.68e-02 -7.85e-02
-2.47e-02 -3.78e-02
-3.14e-02 -1.40e-01
-3.27e-02 -1.40e-01
-8.92e-02 -6.03e-02
-6.18e-02 -2.31e-02
-1.05e-01 -8.98e-03
-4.54e-02 -2.98e-02
-1.02e-01 -1.37e-01
Training
Iterative process

- Forward Pass
- Loss Computation
Training
Iterative process

Forward Pass → Loss Computation → Gradient Pass
Training
Iterative process

Forward Pass → Gradient Pass → Update Weights → Loss Computation → Forward Pass
Training
Forward pass

Input

Training Parameters
Training
Forward pass

Input

9

Output

Training Parameters

-3.40e-02
-1.77e-01
-2.10e-03
-3.53e-02
-1.45e-01
-6.11e-02
-8.27e-02
-9.71e-02
-1.10e-01
-8.75e-02
-5.46e-02
-2.93e-02
-8.77e-03
-8.80e-02
-1.35e-01
-1.97e-01
-4.34e-02
-5.58e-02
-2.96e-02
-1.31e-01
-4.57e-02
-3.39e-02
-4.68e-02
-7.85e-02
-2.47e-02
-3.78e-02
-3.14e-02
-1.40e-01
-3.27e-02
-1.40e-01
-8.92e-02
-6.03e-02
-6.18e-02
-2.31e-02
-1.05e-01
-8.98e-03
-4.54e-02
-2.98e-02
-1.02e-01
-1.37e-01
Training
Forward pass

<table>
<thead>
<tr>
<th>Class</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Input

Training Parameters
Training
Forward pass

Labels

<table>
<thead>
<tr>
<th>Class</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Probabilities

<table>
<thead>
<tr>
<th>Class</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000045</td>
</tr>
<tr>
<td>1</td>
<td>0.000005</td>
</tr>
<tr>
<td>2</td>
<td>0.000000</td>
</tr>
<tr>
<td>3</td>
<td>0.015000</td>
</tr>
<tr>
<td>4</td>
<td>0.185300</td>
</tr>
<tr>
<td>5</td>
<td>0.000026</td>
</tr>
<tr>
<td>6</td>
<td>0.000000</td>
</tr>
<tr>
<td>7</td>
<td>0.000000</td>
</tr>
<tr>
<td>8</td>
<td>0.000126</td>
</tr>
<tr>
<td>9</td>
<td>0.799450</td>
</tr>
</tbody>
</table>

Input

Training Parameters

Class

0
1
2
3
4
5
6
7
8
9
Training
Loss computation

Input

Training Parameters

Class | Label | Probability
--- | --- | ---
0 | 0 | 0.000045
1 | 0 | 0.000005
2 | 0 | 0.000000
3 | 0 | 0.015000
4 | 0 | 0.185300
5 | 0 | 0.000026
6 | 0 | 0.000000
7 | 1 | 0.000000
8 | 0 | 0.000126
9 | 0 | 0.799450
Training
Loss computation

Input

Class | Label | Probability
--- | --- | ---
0 | 0 | 0.000045
1 | 0 | 0.000005
2 | 0 | 0.000000
3 | 0 | 0.015000
4 | 0 | 0.185300
5 | 0 | 0.000026
6 | 0 | 0.000000
7 | 1 | 0.000000
8 | 0 | 0.000126
9 | 0 | 0.799450

Compute loss between predictions and labels

Training Parameters

Loss
Training Parameters

<table>
<thead>
<tr>
<th>Convolution Gradient</th>
<th>ReLu Gradient</th>
<th>Max Pooling Gradient</th>
<th>Fully-Connected Gradient</th>
<th>Fully-Connected</th>
<th>Dropout Gradient</th>
<th>SoftMax Gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5.46e-02</td>
<td>-2.93e-02</td>
<td>-8.77e-03</td>
<td>-8.80e-02</td>
<td>-1.35e-01</td>
<td>-1.97e-01</td>
<td>-4.34e-02</td>
</tr>
<tr>
<td>-8.77e-03</td>
<td>-8.80e-02</td>
<td>-1.35e-01</td>
<td>-1.97e-01</td>
<td>-4.34e-02</td>
<td>-1.97e-01</td>
<td>-2.96e-02</td>
</tr>
<tr>
<td>-1.35e-01</td>
<td>-1.97e-01</td>
<td>-4.34e-02</td>
<td>-1.97e-01</td>
<td>-2.96e-02</td>
<td>-1.97e-01</td>
<td>-4.57e-02</td>
</tr>
<tr>
<td>-4.57e-02</td>
<td>-3.39e-02</td>
<td>-4.68e-02</td>
<td>-7.85e-02</td>
<td>-2.47e-02</td>
<td>-3.78e-02</td>
<td>-3.27e-02</td>
</tr>
<tr>
<td>-4.68e-02</td>
<td>-7.85e-02</td>
<td>-2.47e-02</td>
<td>-3.78e-02</td>
<td>-3.14e-02</td>
<td>-1.40e-01</td>
<td>-3.27e-02</td>
</tr>
<tr>
<td>-2.47e-02</td>
<td>-3.78e-02</td>
<td>-3.14e-02</td>
<td>-1.40e-01</td>
<td>-3.27e-02</td>
<td>-1.40e-01</td>
<td>-8.92e-02</td>
</tr>
<tr>
<td>-3.14e-02</td>
<td>-1.40e-01</td>
<td>-3.27e-02</td>
<td>-1.40e-01</td>
<td>-8.92e-02</td>
<td>-1.40e-01</td>
<td>-6.18e-02</td>
</tr>
<tr>
<td>-3.27e-02</td>
<td>-1.40e-01</td>
<td>-8.92e-02</td>
<td>-1.40e-01</td>
<td>-6.18e-02</td>
<td>-2.31e-02</td>
<td>-1.05e-01</td>
</tr>
<tr>
<td>-8.92e-02</td>
<td>-6.03e-02</td>
<td>-6.18e-02</td>
<td>-2.31e-02</td>
<td>-1.05e-01</td>
<td>-8.98e-03</td>
<td>-4.54e-02</td>
</tr>
<tr>
<td>-6.18e-02</td>
<td>-2.31e-02</td>
<td>-1.05e-01</td>
<td>-8.98e-03</td>
<td>-4.54e-02</td>
<td>-1.37e-01</td>
<td>-1.02e-01</td>
</tr>
<tr>
<td>-1.05e-01</td>
<td>-8.98e-03</td>
<td>-4.54e-02</td>
<td>-2.98e-02</td>
<td>-1.02e-01</td>
<td>-1.37e-01</td>
<td>-1.37e-01</td>
</tr>
<tr>
<td>-4.54e-02</td>
<td>-2.98e-02</td>
<td>-1.02e-01</td>
<td>-1.37e-01</td>
<td>-1.37e-01</td>
<td>-1.37e-01</td>
<td>-1.37e-01</td>
</tr>
</tbody>
</table>

Loss

Convolution | ReLu | Max Pooling | Fully-Connected | Dropout | SoftMax | Loss
---|------|-------------|-----------------|---------|---------|-------

Training Gradient pass
Training
Weights update

Input

Convolution Gradient
ReLu Gradient
Max Pooling Gradient
Fully-Connected Gradient
Dropout Gradient
SoftMax Gradient

Loss
Convolution
ReLu
Max Pooling
Fully-Connected
Dropout
SoftMax
Loss

Training Parameters
Training Parameters

Batch Labels

Loss

Iteration

Training Parameters

Loss

Accuracy
Loss

Training Parameters

Iteration

Loss
Accuracy

Batch Labels

0 4 6

1 2 5

3 7

-3.40e-02 -1.77e-01

-2.10e-03 -3.53e-02

-1.45e-01 -6.11e-02

-8.27e-02 -9.71e-02

-1.10e-01 -8.75e-02

-5.46e-02 -2.93e-02

-8.77e-03 -8.80e-02

-1.35e-01 -1.97e-01

-4.34e-02 -5.58e-02

-2.96e-02 -1.31e-01

-4.57e-02 -3.39e-02

-4.68e-02 -7.85e-02

-2.47e-02 -3.78e-02

-3.14e-02 -1.40e-01

-3.27e-02 -1.40e-01

-8.92e-02 -6.03e-02

-6.18e-02 -2.31e-02

-1.05e-01 -8.98e-03

-4.54e-02 -2.98e-02

-1.02e-01 -1.37e-01

-8.92e-02 -6.03e-02

-6.18e-02 -2.31e-02

-1.05e-01 -8.98e-03

-4.54e-02 -2.98e-02

-1.02e-01 -1.37e-01
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Loss</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Batch**

**Labels**

**Loss**

**Training Parameters**
Training a Neural Network with MPS

Create training graph

Prepare inputs

Specify weights

Execute graph
  • Graph updates weights

Complete training process
Training a Neural Network with MPS

Create training graph

Prepare inputs

Specify weights

Execute graph
  • Graph updates weights

Complete training process
Create Training Graph
Neural Network Graph API

Describe neural network using graph API

- Convolution
- ReLu
- Max Pooling
- Fully Connected
- Dropout
- SoftMax
- Convolution Gradient
- ReLu Gradient
- Max Pooling Gradient
- Fully-Connected Gradient
- Dropout Gradient
- SoftMax Gradient
- Loss
- Image
Create Training Graph
Neural Network Graph API

Describe neural network using graph API

Image nodes—Data

Convolution
ReLu
Max Pooling
Fully Connected
Dropout
SoftMax

Convolution Gradient
ReLu Gradient
Max Pooling Gradient
Fully-Connected Gradient
Dropout Gradient
SoftMax Gradient

Loss
Image
Create Training Graph
Neural Network Graph API

Describe neural network using graph API

Image nodes—Data

Filter nodes—Operations
Create Training Graph

Neural Network Graph API

Describe neural network using graph API

Image nodes—Data

Filter nodes—Operations

```swift
let inputImage = MPSNNImageNode(handle: nil)

let conv = MPSCNNConvolutionNode(source: inputImage,
                                   weights: MyWeights(withFile:"conv.dat"))

let convGradient = conv.gradientFilter(source: previousNode.resultImage)
```
Create Training Graph
Neural Network Graph API

Describe neural network using graph API

Image nodes—Data

Filter nodes—Operations

```swift
let inputImage = MPSNNImageNode(handle: nil)

let conv = MPSCNNConvolutionNode(source: inputImage,
                                 weights: MyWeights(withFile:"conv.dat"))

let convGradient = conv.gradientFilter(source: previousNode.resultImage)
```
Create Training Graph

Neural Network Graph API

Describe neural network using graph API

Image nodes—Data

Filter nodes—Operations

```swift
let inputImage = MPSNNImageNode(handle: nil)

let conv = MPSCNNConvolutionNode(source: inputImage,
                                    weights: MyWeights(withFile: "conv.dat"))

let convGradient = conv.gradientFilter(source: previousNode.resultImage)
```
Describe neural network using graph API

Image nodes—Data

Filter nodes—Operations

```swift
let inputImage = MPSNNImageNode(handle: nil)

let conv = MPSCNNConvolutionNode(source: inputImage,
                                   weights: MyWeights(withFile: "conv.dat"))

let convGradient = conv.gradientFilter(source: previousNode.resultImage)
```
// Example: Create an Inference Graph

func makeInferenceGraph() -> MPSNNImageNode {
    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file: "conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file: "conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file: "fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file: "fc2.dat"))
    return fc2.resultImage
}
// Example: Create an Inference Graph

```swift
func makeInferenceGraph() -> MPSNNImageNode {
    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file: "conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file: "conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file: "fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file: "fc2.dat"))
    return fc2.resultImage
}
```
// Example: Create an Inference Graph

func makeInferenceGraph() -> MPSNNImageNode {

    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file:"conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file:"conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file:"fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file:"fc2.dat"))

    return fc2.resultImage
}

// Example: Create a Training Graph

func makeTrainingGraph() -> MPSNNImageNode {

    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file:"conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file:"conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file:"fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file:"fc2.dat"))
    let loss = MPSCNNLossNode(source: fc2.resultImage, lossDescriptor: lossDescriptor)

    ...
}

// Example: Create a Training Graph

func makeTrainingGraph() -> MPSNNImageNode {

    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file:"conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file:"conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file:"fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file:"fc2.dat"))
    let loss = MPSCNNLossNode(source: fc2.resultImage, lossDescriptor: lossDescriptor)

    ...
}

func makeTrainingGraph() -> MPSNNImageNode {
    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file:"conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file:"conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file:"fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file:"fc2.dat"))
    let loss = MPSCNNLossNode(source: fc2.resultImage, lossDescriptor: lossDescriptor)
    let fc2G = fc2.gradientFilter(source: loss.resultImage)
    let fc1G = fc1.gradientFilter(source: fc2G.resultImage)
    let pool2G = pool2.gradientFilter(source: fc1G.resultImage)
    let conv2G = conv2.gradientFilter(source: pool2G.resultImage)
    let pool1G = pool1.gradientFilter(source: conv2G.resultImage)
    let conv1G = conv1.gradientFilter(source: pool1G.resultImage)
    return conv1G.resultImage
}
func makeTrainingGraph() -> MPSNNImageNode {
    let conv1 = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil), weights: MyWeights(file: "conv1.dat"))
    let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage, filterSize: 2)
    let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage, weights: MyWeights(file: "conv2.dat"))
    let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage, filterSize: 2)
    let fc1 = MPSCNNFullyConnectedNode(source: pool2.resultImage, weights: MyWeights(file: "fc1.dat"))
    let fc2 = MPSCNNFullyConnectedNode(source: fc1.resultImage, weights: MyWeights(file: "fc2.dat"))
    let loss = MPSCNNLossNode(source: fc2.resultImage, lossDescriptor: lossDescriptor)
    let fc2G = fc2.gradientFilter(source: loss.resultImage)
    let fc1G = fc1.gradientFilter(source: fc2G.resultImage)
    let pool2G = pool2.gradientFilter(source: fc1G.resultImage)
    let conv2G = conv2.gradientFilter(source: pool2G.resultImage)
    let pool1G = pool1.gradientFilter(source: conv2G.resultImage)
    let conv1G = conv1.gradientFilter(source: pool1G.resultImage)
    return conv1G.resultImage
}
Graph API Benefits

Very easy to use

Minimizes memory footprint

Fuses and optimizes away graph nodes

Automatically handles padding

Automatically manages state objects
Training a Neural Network with MPS

Create training graph
Prepare inputs
Specify weights
Execute graph
• Graph updates weights
Complete training process
Prepare Inputs

Inputs to the graph
• Batch of source images
• Batch of source states

// Encode a batch of images for training
trainingGraph.encodeBatch(to: commandBuffer!,
sourceImages: [imageBatch],
sourceStates: [statesBatch])
Prepare Inputs

Inputs to the graph
• Batch of source images
• Batch of source states

// Encode a batch of images for training
trainingGraph.encodeBatch(to: commandBuffer!,
    sourceImages: [imageBatch],
    sourceStates: [statesBatch])
Batches are arrays of images or states

```swift
// Create an input image
let inputImage0 = MPSImage(texture: texture0, featureChannels: 4)
...

// Create a batch of input images
var imageBatch : [MPSImage] = []
imageBatch.append(inputImage0)
...```
Batches

Batches are arrays of images or states

**MPSImageBatch, MPSStateBatch**

```swift
// Create an input image
let inputImage0 = MPSImage(texture: texture0, featureChannels: 4)

// Create a batch of input images
var imageBatch : [MPSImage] = []
imageBatch.append(inputImage0)
```
Batches

Batches are arrays of images or states

**MPSImageBatch, MPSStateBatch**

```swift
// Create an input image
let inputImage0 = MPSImage(texture: texture0, featureChannels: 4)
...

// Create a batch of input images
var imageBatch : [MPSImage] = []
imageBatch.append(inputImage0)
...```
States

MPSState passes state of forward node to gradient node

Graph manages all states
Dropout
keepProbability = 0.7

Dropout Gradient
keepProbability = 0.7
Dropout
keepProbability = 0.7

Dropout Gradient
keepProbability = 0.7
Dropout
keepProbability = 0.7

Dropout Gradient
keepProbability = 0.7
Dropout
keepProbability = 0.7

Input

Dropout Gradient
keepProbability = 0.7

Output

State

Input

Input Gradient

Output

State
Dropout
keepProbability = 0.7

Dropout Gradient
keepProbability = 0.7

Input Gradient
Output Gradient
State
// Create one label for Loss computation

let labelData = . . . // Load label data

let labelDesc = MPSCNNLossDataDescriptor(data: Data(bytes: labelData),
    layout: .featureChannelsxHeightxWidth,
    size: MTLSize(width: 1, height: 1, depth: 10))!

let cnnLabel = MPSCNNLossLabels(device: device,
    labelsDescriptor: labelDesc)
// Create one label for Loss computation
let labelData = . . . // Load label data

let labelDesc = MPSCNNLossDataDescriptor(data: Data(bytes: labelData),
                                         layout: .featureChannelsxHeightxWidth,
                                         size: MTLSize(width: 1, height: 1, depth: 10))!

let cnnLabel = MPSCNNLossLabels(device: device,
                                 labelsDescriptor: labelDesc)
// Create one label for Loss computation
let labelData = . . . // Load label data
let labelDesc = MPSCNNLossDataDescriptor(data: Data(bytes: labelData),
    layout: .featureChannelsXHeightXWidth,
    size: MTLSize(width: 1, height: 1, depth: 10))!

let cnnLabel = MPSCNNLossLabels(device: device,
    labelsDescriptor: labelDesc)
Training a Neural Network with MPS

Create training graph
Prepare inputs
Specify weights
Execute graph
  • Graph updates weights
Complete training process
**Data Source Providers**

Convolution

Fully Connected

Batch normalization

Instance normalization

```
// User implements MPSConvolutionDataSource protocol
let conv = MPSConvolutionNode(source: MPSImageNode(handle: nil),
weights: MyWeights(withFile: "conv.dat"))
```
Data Source Providers

Convolution

Fully Connected

Batch normalization

Instance normalization

// User implements MPSCNNConvolutionDataSource protocol
let conv = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil),
weights: MyWeights(withFile:"conv.dat"))
Data Source Providers

Convolution

Fully Connected

Batch normalization

Instance normalization

// User implements MPSCNNConvolutionDataSource protocol
let conv = MPSCNNConvolutionNode(source: MPSNNImageNode(handle: nil),
                                  weights: MyWeights(withFile: "conv.dat"))
Data Source Providers

Just-in-time loading and purging of weights data

Minimize memory footprint
Data Source Providers

Just-in-time loading and purging of weights data

Minimize memory footprint

class MyWeights: NSObject, MPSCNNConvolutionDataSource {
    init(file: String) {...} // Initialize the data source object
    func load() -> Bool {...} // Graph calls load when weights for this layer are needed
    func descriptor() -> MPSCNNConvolutionDescriptor {...}
    func weights() -> UnsafeMutableRawPointer {...}
    func purge() {...} // When purge is called, you can release weights
}
Data Source Providers

Just-in-time loading and purging of weights data

Minimize memory footprint

class MyWeights: NSObject, MPSCNNConvolutionDataSource {
    init(file: String) {...} // Initialize the data source object
    func load() -> Bool {...} // Graph calls load when weights for this layer are needed
    func descriptor() -> MPSCNNConvolutionDescriptor {...}
    func weights() -> UnsafeMutableRawPointer {...}
    func purge() {...} // When purge is called, you can release weights
}
Data Source Providers

Just-in-time loading and purging of weights data

Minimize memory footprint

class MyWeights: NSObject, MPSCNNConvolutionDataSource {
    init(file: String) {...} // Initialize the data source object
    func load() -> Bool {...} // Graph calls load when weights for this layer are needed
    func descriptor() -> MPSCNNConvolutionDescriptor {...}
    func weights() -> UnsafeMutableRawPointer {...}
    func purge() {...} // When purge is called, you can release weights
}
Training a Neural Network with MPS

Create training graph
Prepare inputs
Specify weights
Execute graph
  • Graph updates weights
Complete training process
// Example: Prepare to Execute Graph on the GPU

// Metal setup
let device = MTLCreateSystemDefaultDevice()

// Initialize graph
let trainingGraph = MPSNNGraph(device: device!, resultImage: makeTrainingGraph())

// Prepare inputs
...

// Train network on the GPU
...

// Example: Prepare to Execute Graph on the GPU

// Metal setup
let device = MTLCreateSystemDefaultDevice()

// Initialize graph
let trainingGraph = MPSNNGraph(device: device!, resultImage: makeTrainingGraph())

// Prepare inputs
...

// Train network on the GPU
...

// Example: Prepare to Execute Graph on the GPU

// Metal setup
let device = MTLCreateSystemDefaultDevice()

// Initialize graph
let trainingGraph = MPSNNGraph(device: device!, resultImage: makeTrainingGraph())

// Prepare inputs
...

// Train network on the GPU
...

// Example: Prepare to Execute Graph on the GPU

// Metal setup
let device = MTLCreateSystemDefaultDevice()

// Initialize graph
let trainingGraph = MPSNNGraph(device: device!, resultImage: makeTrainingGraph())

// Prepare inputs
...

// Train network on the GPU
...

// Example: Prepare to Execute Graph on the GPU

// Metal setup
let device = MTLCreateSystemDefaultDevice()

// Initialize graph
let trainingGraph = MPSNNGraph(device: device!, resultImage: makeTrainingGraph())

// Prepare inputs
...

// Train network on the GPU
...

// Example: Execute Graph in a Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
    ...
// Example: Execute Graph in a Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
    ...
// Example: Execute Graph in a Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
    ...
let doubleBufferSemaphore = DispatchSemaphore(value: 2)

func trainingIteration(_ iter: Int) -> MTLCommandBuffer {
    doubleBufferSemaphore.wait(timeout: .distantFuture)
    let commandBuffer = commandQueue.makeCommandBuffer()!
    // Encode a batch of images for training
    var outputBatch = trainingGraph.encodeBatch(to: commandBuffer,
                                              sourceImages: [dataset.nextImageBatch(iter)],
                                              sourceStates: [dataset.nextLabelsBatch(iter)])

    commandBuffer.addCompletedHandler { commandBuffer in
        // Callback is called when GPU is done executing the graph (outputBatch is ready)
        doubleBufferSemaphore.signal()
    }

    commandBuffer.commit()
    return commandBuffer
}
let doubleBufferSemaphore = DispatchSemaphore(value: 2)

func trainingIteration(_ iter: Int) -> MTLCommandBuffer {
    doubleBufferSemaphore.wait(timeout: .distantFuture)
    let commandBuffer = commandQueue.makeCommandBuffer()!

    var outputBatch = trainingGraph.encodeBatch(to: commandBuffer,
        sourceImages: [dataset.nextImageBatch(iter)],
        sourceStates: [dataset.nextLabelsBatch(iter)])

    commandBuffer.addCompletedHandler { commandBuffer in
        doubleBufferSemaphore.signal()
    }
    commandBuffer.commit()
    return commandBuffer
}
let doubleBufferSemaphore = DispatchSemaphore(value: 2)

func trainingIteration(_ iter: Int) -> MTLCommandBuffer {
    doubleBufferSemaphore.wait(timeout: .distantFuture)
    let commandBuffer = commandQueue.makeCommandBuffer()!
    // Encode a batch of images for training
    var outputBatch = trainingGraph.encodeBatch(to: commandBuffer,
                                              sourceImages: [dataset.nextImageBatch(iter)],
                                              sourceStates: [dataset.nextLabelsBatch(iter)])
    commandBuffer.addCompletedHandler {
        commandBuffer in
        // Callback is called when GPU is done executing the graph (outputBatch is ready)
        doubleBufferSemaphore.signal()
    }
    commandBuffer.commit()
    return commandBuffer
}
// Example: Execute Graph in a Training Loop with Double Buffering

let doubleBufferSemaphore = DispatchQueue semaphore(value: 2)

func trainingIteration(_ iter: Int) -> MTLCommandBuffer {
    doubleBufferSemaphore.wait(timeout: .distantFuture)
    let commandBuffer = commandQueue.makeCommandBuffer()!
    // Encode a batch of images for training
    var outputBatch = trainingGraph.encodeBatch(to: commandBuffer,
                                              sourceImages: [dataset.nextImageBatch(iter)],
                                              sourceStates: [dataset.nextLabelsBatch(iter)])

    commandBuffer.addCompletedHandler { commandBuffer in
        // Callback is called when GPU is done executing the graph (outputBatch is ready)
        doubleBufferSemaphore.signal()
    }

    commandBuffer.commit()
    return commandBuffer
}
let doubleBufferSemaphore = DispatchSemaphore(value: 2)

func trainingIteration(_ iter: Int) -> MTLCommandBuffer {
    doubleBufferSemaphore.wait(timeout: .distantFuture)
    let commandBuffer = commandQueue.makeCommandBuffer()!
    // Encode a batch of images for training
    var outputBatch = trainingGraph.encodeBatch(to: commandBuffer, sourceImages: [dataset.nextImageBatch(iter)], sourceStates: [dataset.nextLabelsBatch(iter)])
    commandBuffer.addCompletedHandler { commandBuffer in
        // Callback is called when GPU is done executing the graph (outputBatch is ready)
        doubleBufferSemaphore.signal()
    }
    commandBuffer.commit()
    return commandBuffer
}
Training a Neural Network with MPS

Create training graph
Prepare inputs
Specify weights
Execute graph
  • Graph updates weights
Complete training process
Implement optional update method on Data Source Provider

Graph calls update method automatically

class MyWeights: NSObject, MPSCNNConvolutionDataSource {
    . . .
    func update(with commandBuffer: MTLCommandBuffer,
                gradientState: MPSCNNConvolutionGradientState,
                sourceState: MPSCNNConvolutionWeightsAndBiasesState) ->
        MPSCNNConvolutionWeightsAndBiasesState? { ... } // GPU
Implement optional update method on Data Source Provider

Graph calls update method automatically

class MyWeights: NSObject, MPSCNNConvolutionDataSource {
  ...

  func update(with commandBuffer: MTLCommandBuffer,
              gradientState: MPSCNNConvolutionGradientState,
              sourceState: MPSCNNConvolutionWeightsAndBiasesState) ->
  MPSCNNConvolutionWeightsAndBiasesState? {
    ... // GPU
  }
}
Describe how to take update step on training parameters

Used in update method of Data Source Provider

Variants

- MPSNNOptimizerAdam
- MPSNNOptimizerStochasticGradientDescent
- MPSNNOptimizerRMSProp
- Custom
// Example: Use an Optimizer to Update Weights

// Add an optimizer to the init method of a Data Source Provider

init(kernelWidth: Int,
     kernelHeight: Int,
     inputFeatureChannels: Int,
     outputFeatureChannels: Int) {
  let length = kernelWidth * kernelHeight * inputFeatureChannels * outputFeatureChannels

  self.optimizer = MPSNNOptimizerStochasticGradientDescent(device: device,
                                                            learningRate: 0.01)
}

// Example: Use an Optimizer to Update Weights

// Add an optimizer to the init method of a Data Source Provider

init(kernelWidth: Int,
     kernelHeight: Int,
     inputFeatureChannels: Int,
     outputFeatureChannels: Int) {
    let length = kernelWidth * kernelHeight * inputFeatureChannels * outputFeatureChannels

    self.optimizer = MPSNNOptimizerStochasticGradientDescent(device: device,
                                                            learningRate: 0.01)
}
func update(with commandBuffer: MTLCommandBuffer,
    gradientState: MPSCNNConvolutionGradientState,
    sourceState: MPSCNNConvolutionWeightsAndBiasesState)
    -> MPSCNNConvolutionWeightsAndBiasesState? {

    optimizer.encode(commandBuffer: commandBuffer,
        convolutionGradientState: gradientState,
        convolutionSourceState: sourceState,
        inputMomentumVectors: nil,
        resultState: sourceState)

    return sourceState
}
// Example: Use an Optimizer to Update Weights

func update(with commandBuffer: MTLCommandBuffer,
    gradientState: MPSCNNConvolutionGradientState,
    sourceState: MPSCNNConvolutionWeightsAndBiasesState)
    -> MPSCNNConvolutionWeightsAndBiasesState? {

    optimizer.encode(commandBuffer: commandBuffer,
        convolutionGradientState: gradientState,
        convolutionSourceState: sourceState,
        inputMomentumVectors: nil,
        resultState: sourceState)

    return sourceState
}
// Example: Use an Optimizer to Update Weights

func update(with commandBuffer: MTLCommandBuffer,
            gradientState: MPSCNNConvolutionGradientState,
            sourceState: MPSCNNConvolutionWeightsAndBiasesState)
            -> MPSCNNConvolutionWeightsAndBiasesState? {

    optimizer.encode(commandBuffer: commandBuffer,
                     convolutionGradientState: gradientState,
                     convolutionSourceState: sourceState,
                     inputMomentumVectors: nil,
                     resultState: sourceState)

    return sourceState

}
// Example: Use an Optimizer to Update Weights

func update(with commandBuffer: MTLCommandBuffer,
            gradientState: MPSCNNConvolutionGradientState,
            sourceState: MPSCNNConvolutionWeightsAndBiasesState)
-> MPSCNNConvolutionWeightsAndBiasesState? {

    optimizer.encode(commandBuffer: commandBuffer,
                     convolutionGradientState: gradientState,
                     convolutionSourceState: sourceState,
                     inputMomentumVectors: nil,
                     resultState: sourceState)

    return sourceState

}
Training a Neural Network with MPS

Create training graph
Prepare inputs
Specify weights
Execute graph
• Graph updates weights
Complete training process
// Example: Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
}

// Optional steps after each epoch:
latestCommandBuffer.waitUntilCompleted()
// Run inference with current trained parameters on a test set
// Optionally, stop training when the network reaches an acceptable level of accuracy
}
// Example: Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
}

// Optional steps after each epoch:
latestCommandBuffer.waitUntilCompleted()

// Run inference with current trained parameters on a test set
// Optionally, stop training when the network reaches an acceptable level of accuracy
}
// Example: Training Loop with Double Buffering

let latestCommandBuffer = nil

// NUM_EPOCHS is the number of times we iterate over an entire dataset
// NUM_ITERATIONS_PER_EPOCH is the number of images in a dataset, divided by batch size
for i in 0..<NUM_EPOCHS {
    for j in 0..<NUM_ITERATIONS_PER_EPOCH {
        latestCommandBuffer = trainingIteration(i * NUM_ITERATIONS_PER_EPOCH + j);
    }
}

// Optional steps after each epoch:
latestCommandBuffer.waitUntilCompleted()

// Run inference with current trained parameters on a test set
// Optionally, stop training when the network reaches an acceptable level of accuracy
}
Demo

Object detection training using Turi Create with MPS
RNN Training
CNN
One-to-one

One image
CNN
One-to-one

One image

One set of probabilities

cat
dog
grass
...

CNN

One set of probabilities
RNN
Sequences: one-to-many

One image

CNN
RNN

Sequences: one-to-many

One image

CNN

One set of probabilities

RNN

Text:

A dog is laying on the grass next to a cat

Sequence of words
A dog is laying on the grass next to a cat

Sequence of words

RNN
Sequences: many-to-many
RNN
Sequences: many-to-many

A dog is laying on the grass next to a cat

Sequence of words

RNN

Собака лежит на траве рядом с котом
Koira makaa ruohikolla kissan vieressä

Sequence of words
Recurrent Neural Networks

Variants for inference
• Single Gate
• Long Short-Term Memory (LSTM)
• Gated Recurrent Unit (GRU)
• Minimally Gated Unit (MGU)
Recurrent Neural Networks

Variants for inference and training

• Single Gate
• Long Short-Term Memory (LSTM)
• Gated Recurrent Unit (GRU)
• Minimally Gated Unit (MGU)
Activity Classifier

Network overview

*Graphical representation of motion-sensory data for the purpose of demonstration, not actual data
Inference

1D images with 2000 pixels (samples in time)

Input

LSTM

Convolution
ReLu
Fully Connected
Dropout
Batch Normalization
SoftMax
LSTM
LSTM

20x

Activity Predictions

Output
Inference

1D images with 2000 pixels (samples in time)

1D Convolution

“Compresses” input to 20 samples

LSTM

20x

Activity Predictions

Convolution
ReLu
Fully Connected
Dropout
Batch Normalization
SoftMax
LSTM
LSTM
Batch Normalization
Fully Connected
Dropout
ReLu
Convolution

Input

Output
Inference

Input

1D images with 2000 pixels (samples in time)

“Compresses” input to 20 samples

Sequence of length 20

“Compresses” input to 20 samples

20x

LSTM

Activity Predictions

Output

Convolution
ReLu
Fully Connected
Dropout
Batch Normalization
SoftMax
LSTM
LSTM
LSTM

Compresses input to 20 samples

Sequence of length 20

Input

Output
Inference

1D images with 2000 pixels (samples in time)

“Compresses” input to 20 samples

Sequence of length 20

Refine higher level features

Activity Predictions

Input

Output
Training

Images → LSTM → Activity Predictions

Labels

Convolution → ReLu → Fully Connected → Dropout → Batch Normalization → SoftMax → LSTM → LSTM Gradient

Convolution Gradient → ReLu Gradient → Fully Connected Gradient → Dropout Gradient → Batch Normalization Gradient → SoftMax Gradient
**Training**

Images → LSTM → Convolution → ReLu → Fully Connected → Dropout → Batch Normalization → SoftMax → LSTM → SoftMax Gradient → Batch Normalization Gradient → Dropout Gradient → Fully Connected Gradient → ReLu Gradient → Convolution Gradient → Loss

Activity Predictions → Labels
Training
Weights update

Images → LSTM (20x) → LSTM (20x) → Activity Predictions

Convolution, ReLu, Fully Connected, Dropout, Batch Normalization, SoftMax, LSTM, LSTM Gradient

Convolution Gradient, ReLu Gradient, Fully Connected Gradient, Dropout Gradient, Batch Normalization Gradient, SoftMax Gradient, LSTM Gradient

Labels
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescr iptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMut ableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
                                               trainableWeights: trainingWeights)
trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescriptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMutableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
trainableWeights: trainingWeights)

trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescr iptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMutableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
trainableWeights: trainingWeights)
trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescrptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMutableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
trainableWeights: trainingWeights)
trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescr iptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMutableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
trainableWeights: trainingWeights)

trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Create a LSTM Layer for Training

// Create a LSTM layer descriptor
// Initialize descriptor with initial training parameters, using data source providers
// Descriptor setup is exactly the same as for inference
let descriptor = MPSLSTMDescriptor()
descriptor.useFloat32Weights = true

// Create a LSTM layer for training
let trainingWeights: NSMutableArray = [] // matrices to hold training parameters
let weightGradients: NSMutableArray = [] // matrices to hold gradients
let trainingLayer = MPSRNNMatrixTrainingLayer(device: device!, rnnDescriptor: descriptor,
                                        trainableWeights: trainingWeights)

trainingLayer.createWeightGradientMatrices(weightGradients, dataType: .float32)
// Example: Prepare Inputs and Outputs for Training a LSTM

// Create 20 matrices to hold input and output sequences for forward and gradient passes
var inputSequence: [MPSMatrix] = []
var outputSequence: [MPSMatrix] = []
var inputGradientSequence: [MPSMatrix] = []
var outputGradientSequence: [MPSMatrix] = []

for i in 0..<20 {
    // Matrix size is (1, inputSize), inputSize is number of columns
    inputSequence.append(MPSMatrix(...))
    // Initialize matrices for the other sequences
    . . .
}

// Example: Prepare Inputs and Outputs for Training a LSTM

// Create 20 matrices to hold input and output sequences for forward and gradient passes
var inputSequence: [MPSMatrix] = []
var outputSequence: [MPSMatrix] = []
var inputGradientSequence: [MPSMatrix] = []
var outputGradientSequence: [MPSMatrix] = []

for i in 0..<20 {
    // Matrix size is (1, inputSize), inputSize is number of columns
    inputSequence.append(MPSMatrix(...))
    // Initialize matrices for the other sequences
    ...
}
// Example: Prepare Inputs and Outputs for Training a LSTM

// Create 20 matrices to hold input and output sequences for forward and gradient passes
var inputSequence: [MPSMatrix] = []
var outputSequence: [MPSMatrix] = []
var inputGradientSequence: [MPSMatrix] = []
var outputGradientSequence: [MPSMatrix] = []

for i in 0..<20 {
    // Matrix size is (1, inputSize), inputSize is number of columns
    inputSequence.append(MPSMatrix(...))
    // Initialize matrices for the other sequences
    ...
}
// Example: Train Activity Classifier Network with MPS

// Run CNN kernels in forward pass

let trainingStates: NS MUT ableArray = [] // States pass data from forward to gradient pass
// Run sequence of 20 matrices through LSTM training layer (forward pass)
trainingLayer.encodeForwardSequence(commandBuffer: commandBuffer,
sourceMatrices: inputSequence,
destinationMatrices: outputSequence,
trainingStates: trainingStates,
weights: trainingWeights)

// Run additional CNN kernels in forward pass
// Compute Loss
// Example: Train Activity Classifier Network with MPS

// Run CNN kernels in forward pass

let trainingStates: [Any] = [] // States pass data from forward to gradient pass

// Run sequence of 20 matrices through LSTM training layer (forward pass)
trainingLayer.encodeForwardSequence(commandBuffer: commandBuffer,
                                   sourceMatrices: inputSequence,
                                   destinationMatrices: outputSequence,
                                   trainingStates: trainingStates,
                                   weights: trainingWeights)

// Run additional CNN kernels in forward pass

// Compute Loss
// Example: Train Activity Classifier Network with MPS, cont.

// Run CNN kernels in gradient pass

// Run sequence of 20 matrices through LSTM training layer (gradient pass)
trainingLayer.encodeGradientSequence(commandBuffer: commandBuffer,
    forwardSources: forwardSources,
    sourceGradients: inputGradientSequence,
    destinationGradients: outputGradientSequence,
    weightGradients: weightGradients,
    trainingStates: trainingStates,
    weights: trainingWeights)

// Run additional CNN kernels in gradient pass

// Use MPSNNOptimizer to update trainingWeights using weightGradients

// Submit work to the GPU
// Example: Train Activity Classifier Network with MPS, cont.

// Run CNN kernels in gradient pass

// Run sequence of 20 matrices through LSTM training layer (gradient pass)
trainingLayer.encodeGradientSequence{
  commandBuffer: commandBuffer,
  forwardSources: forwardSources,
  sourceGradients: inputGradientSequence,
  destinationGradients: outputGradientSequence,
  weightGradients: weightGradients,
  trainingStates: trainingStates,
  weights: trainingWeights
}

// Run additional CNN kernels in gradient pass
// Use MPSNNOptimizer to update trainingWeights using weightGradients
// Submit work to the GPU
// Example: Train Activity Classifier Network with MPS, cont.

// Run CNN kernels in gradient pass

// Run sequence of 20 matrices through LSTM training layer (gradient pass)
trainingLayer.encodeGradientSequence(commandBuffer: commandBuffer,
                                      forwardSources: forwardSources,
                                      sourceGradients: inputGradientSequence,
                                      destinationGradients: outputGradientSequence,
                                      weightGradients: weightGradients,
                                      trainingStates: trainingStates,
                                      weights: trainingWeights)

// Run additional CNN kernels in gradient pass

// Use MPSNNOptimizer to update trainingWeights using weightGradients

// Submit work to the GPU
Data Converters
Image to/from matrix

Input Output
LSTM
20x
Image ImageMatrix
Convolution
ReLu
Fully Connected
Dropout
Batch Normalization
SoftMax
LSTM
LSTM
SoftMax
Batch Normalization
Dropout
Fully Connected
ReLu
Convolution
Input
Image
Matrix
Output
Image
Data Converters
Image to/from matrix

let imageToMatrix = MPSImageCopyToMatrix(device: device, dataLayout: .featureChannelsxHeightxWidth)
imageToMatrix.encodeBatch(commandBuffer: commandBuffer, sourceImages: [image1, image2],
destinationMatrix: matrix)
let imageToMatrix = MPSImageCopyToMatrix(device: device, dataLayout: .featureChannelsxHeightxWidth)
imageToMatrix.encodeBatch(commandBuffer: commandBuffer, sourceImages: [image1, image2], destinationMatrix: matrix)
Data Converters
Image to/from matrix

```
let imageToMatrix = MPSImageCopyToMatrix(device: device, dataLayout: .featureChannelsxHeightxWidth)
imageToMatrix.encodeBatch(commandBuffer: commandBuffer, sourceImages: [image1, image2],
destinationMatrix: matrix)
```
let matrixToImage = MPSMatrixCopyToImage(device: device, dataLayout: .featureChannelsxHeightxWidth)

matrixToImage.encodeBatch(commandBuffer: commandBuffer, sourceMatrix: matrix, destinationImages: [image1, image2])
Data Converters

Image to/from matrix

```swift
let matrixToImage = MPSMatrixCopyToImage(device: device, dataLayout: .featureChannelsxHeightxWidth)
matrixToImage.encodeBatch(commandBuffer: commandBuffer, sourceMatrix: matrix,
                          destinationImages: [image1, image2])
```
```
let matrixToImage = MPSMatrixCopyToImage(device: device, dataLayout: .featureChannelsxHeightxWidth)
matrixToImage.encodeBatch(commandBuffer: commandBuffer, sourceMatrix: matrix,
    destinationImages: [image1, image2])
```
Demo

Object classification training using TensorFlow with MPS
Demo

Object classification training using TensorFlow with MPS

Summary

FP16 accumulation for inference

GPU-accelerated primitives
  • For training neural networks
  • Optimized for iOS and macOS
  • Neural Network Graph API for ease of use
More Information

https://developer.apple.com/wwdc18/609