Training DNNs on iOS with Metal Performance Shaders Graph

NSSpain XI

Noah Martin | Co-Founder, Emerge Tools

Training in Data Centers

$10k+ Server GPUs

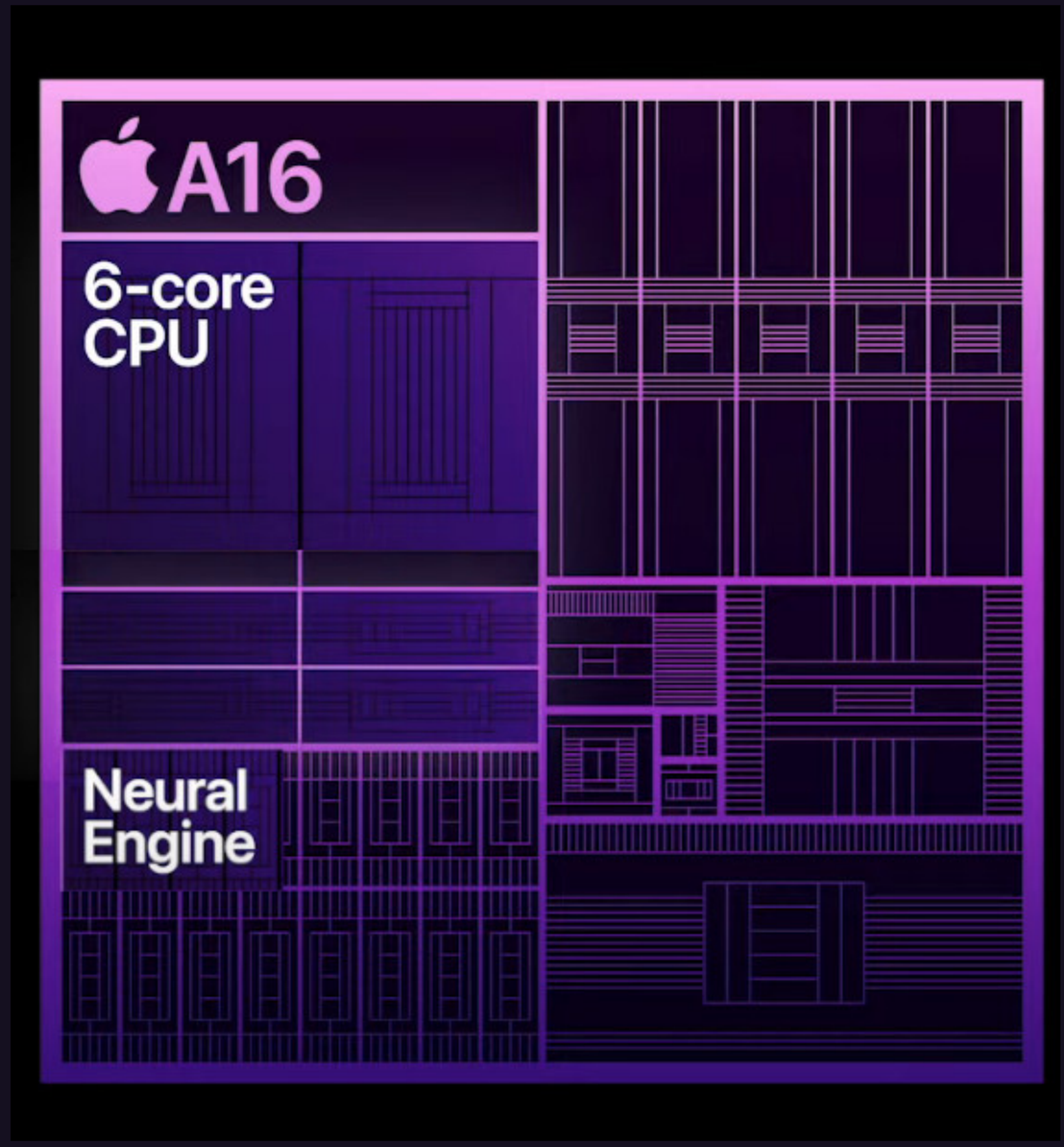
2k A100 NVIDIA GPUs to train Facebook's LLaMa

$30M

For all GPUs in Cluster

Using iPhone GPUs

- Not only iPhones, also Mac and iPad
- Comparison with $5k Nvidia A30
  - 1/6 FLOPS = floating point operations/second
  - 1/4 memory
  - 1/20 memory bandwidth



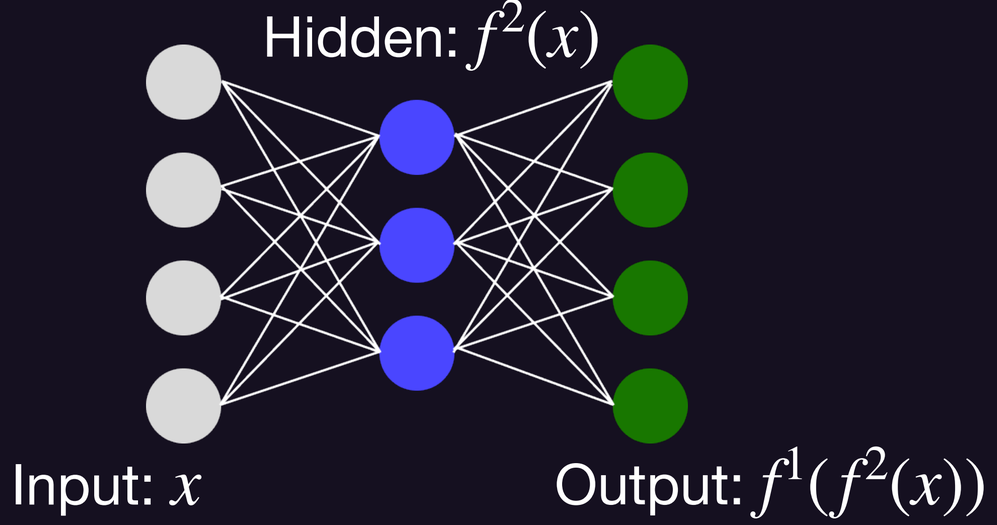
Roadmap

- Background on neural networks
- Building ResNet-50 in TensorFlow
- Building ResNet-50 in Metal Performance Shaders Graph
- Demo training on iOS

DNN Fundamentals

Setup

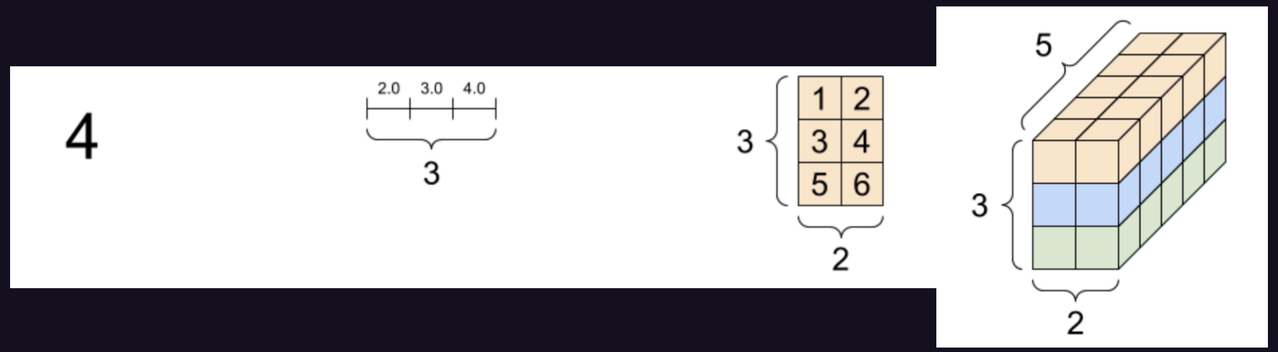
- Goal: Find function f^* to map input x to output y. $y = f^*(x)$
- Strategy: Define $f(x) = f^1(f^2(x))$ to approximate $f^*(x)$.
- Create a loss function $L(f(x), y)$ to be the amount of error in our approximation



DNN Fundamentals

Tensors

- Each layer operates on a tensor, a multidimensional array



Scalar Vector Matrix Tensor

DNN Fundamentals

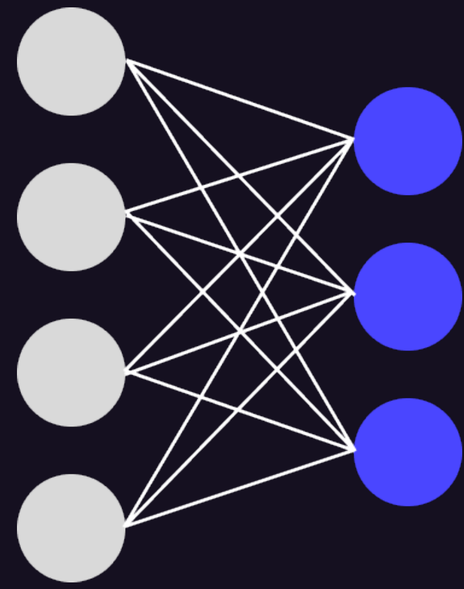
Calc Review

- Each layer, $f^i(x)$, is defined by a weight tensor, θ^i
- Find θ^i to minimize the loss function
- Iteratively update θ^i in the direction opposite of the gradient.

DNN Fundamentals

Common Layers

- Fully Connected
- (Max/Min/Avg) Pooling
- Convolution
- Activation
- Batch Normalization

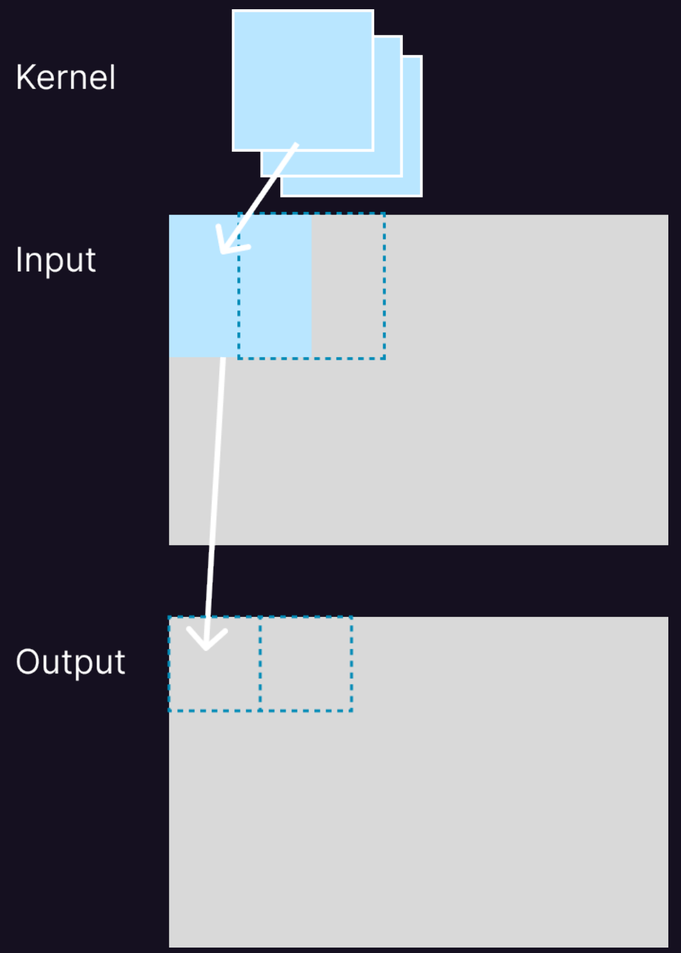


$$Output = Input \cdot W + B$$

DNN Fundamentals

Common Layers

- Fully Connected
- (Max/Min/Avg) Pooling
- Convolution
- Activation
- Batch Normalization

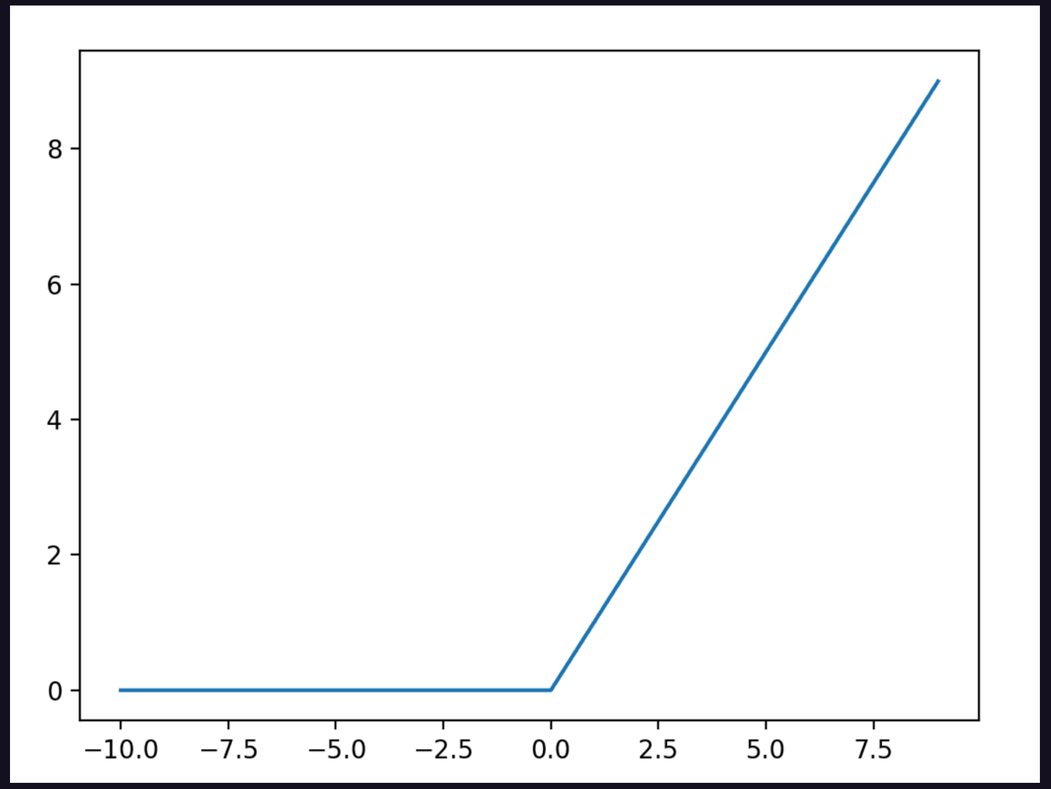


DNN Fundamentals

Common Layers

- Fully Connected
- (Max/Min/Avg) Pooling
- Convolution
- Activation
- Batch Normalization

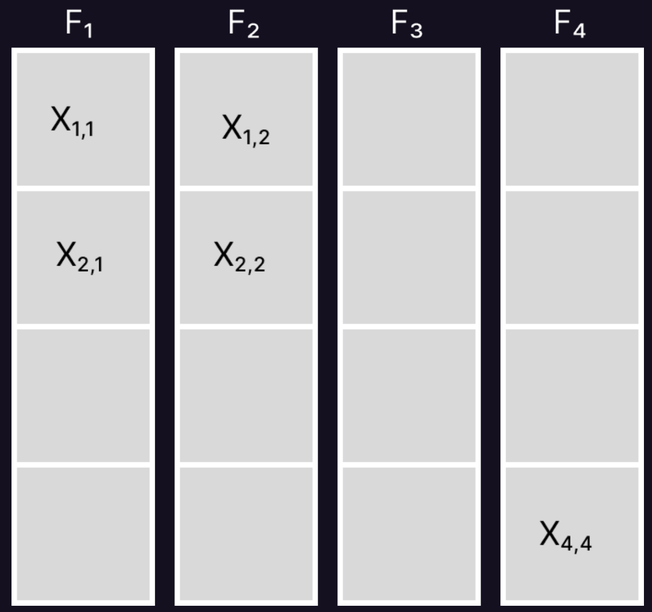
ReLU



DNN Fundamentals

Common Layers

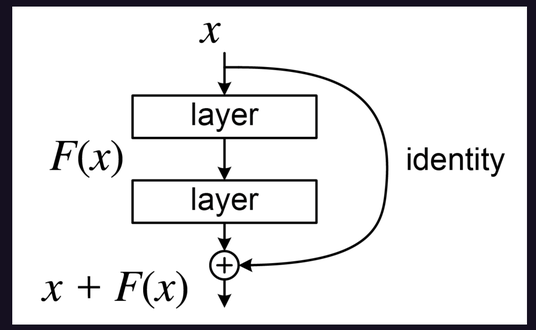
- Fully Connected
- (Max/Min/Avg) Pooling
- Convolution
- Activation
- Batch Normalization

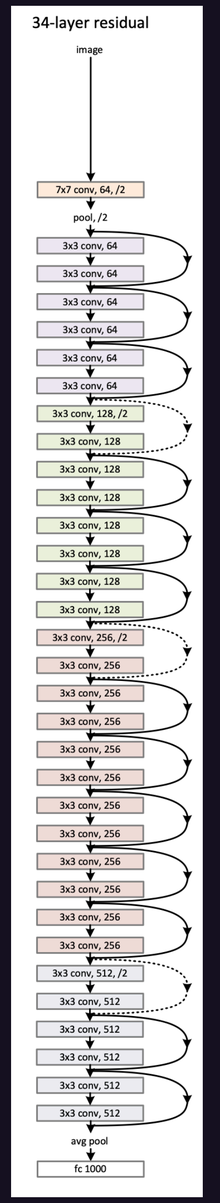


$$X_{i,y} = \frac{X_{i,y} - Mean(y)}{StdDev(y)}$$

ResNet-50 in TensorFlow

- 50 layers, with input image 32x32x3, 10 classes, used for object classification



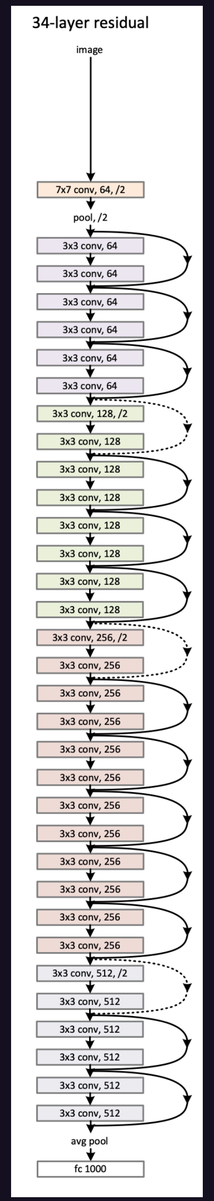



```
def res_identity(x, filters):
  x_skip = x
  f1, f2 = filters

  x = Conv2D(f1, kernel_size=(1, 1), strides=(1, 1), padding='valid', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation(activations.relu)(x)

  x = Conv2D(f1, kernel_size=(3, 3), strides=(1, 1), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation(activations.relu)(x)

  x = Conv2D(f2, kernel_size=(1, 1), strides=(1, 1), padding='valid', use_bias=False)(x)
  x = BatchNormalization()(x)

  x = Add()([x, x_skip])
  x = Activation(activations.relu)(x)

  return x
```

ResNet-50 in TensorFlow

- Prepare inputs and first few layers

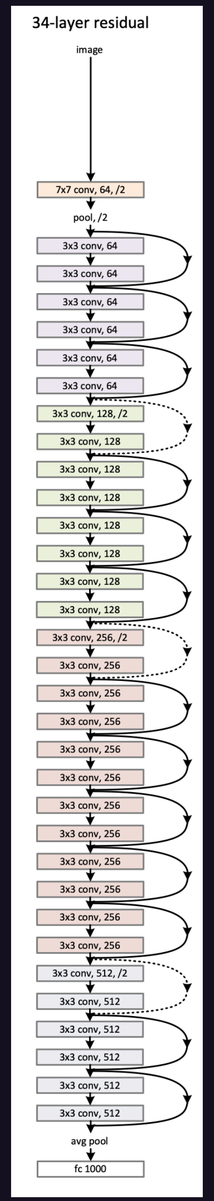
```
input = Input(shape=(train_im.shape[1], train_im.shape[2], train_im.shape[3]), batch_size=batch_size)
x = Conv2D(2, kernel_size=(7, 7), strides=(2, 2), use_bias=False)(input)
x = BatchNormalization()(x)
x = Activation(activations.relu)(x)
x = MaxPooling2D((3, 3), strides=(2, 2))(x)
```



ResNet-50 in TensorFlow

- Add residual layers

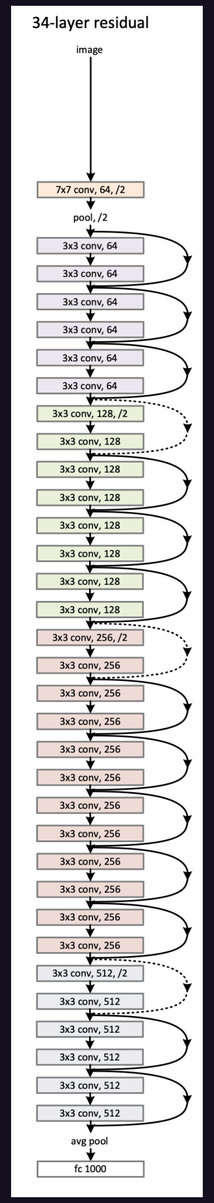
```
x = res_conv(x, s=1, filters=(64, 256))
x = res_identity(x, filters=(64, 256))
x = res_identity(x, filters=(64, 256))
x = res_conv(x, s=2, filters=(128, 512))
x = res_identity(x, filters=(128, 512))
x = res_identity(x, filters=(128, 512))
x = res_identity(x, filters=(128, 512))
x = res_conv(x, s=2, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_identity(x, filters=(256, 1024))
x = res_conv(x, s=2, filters=(512, 2048))
x = res_identity(x, filters=(512, 2048))
x = res_identity(x, filters=(512, 2048))
```



ResNet-50 in TensorFlow

- Finalize outputs and start training

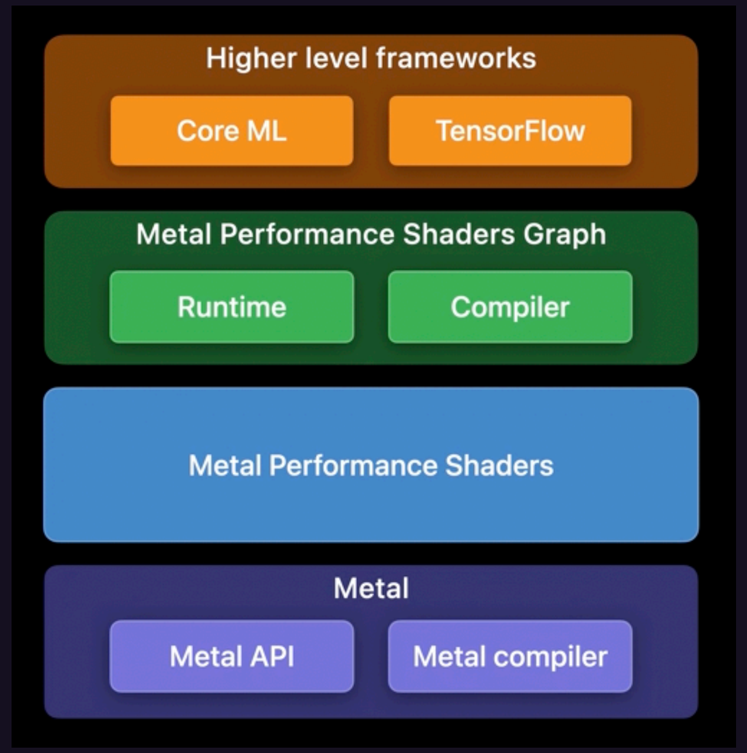
```
x = AveragePooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
x = Dense(10, activation='softmax')(x)

model = Model(inputs=input, outputs=x, name='Resnet50')
opt = tf.keras.optimizers.legacy.SGD()
model.compile(
  loss=tf.keras.losses.CategoricalCrossentropy(reduction=tf.keras.losses.Reduction.SUM_OVER_BATCH_SIZE),
  optimizer=opt,
  metrics=['acc'])
model.fit(train_set_data, epochs=5, steps_per_epoch=train_im.shape[0]/batch_size)
```



Metal Performance Shaders Graph

MPSGraph

- Graph based API to arrange compute operations on the GPU
- More control over GPU operations than CoreML
- Includes optimizations such as layer fusing
- TensorFlow on Mac is built on MPSGraph
- No support for Neural Engine
- We'll use this framework to build and train a model on device, similar to TensorFlow



MPSGraph

- Entire graph managed by MPSGraph object
- Create layers:
  - convolution2D(
    _ source: MPSGraphTensor,
    weights: MPSGraphTensor,
    descriptor: MPSGraphConvolution2DOpDescriptor) -> MPSGraphTensor
- Load data:
  - variable(
    with data: Data,
    shape: [NSNumber],
    dataType: MPSDataType) -> MPSGraphTensor

Create Graph

Setup inputs

```
import MetalPerformanceShadersGraph

let batchSize = 64 as NSNumber
let graph = MPSGraph()
let weightTensors = [MPSGraphTensor]()

let inputPlaceholder = graph.placeholder(shape: [batchSize, 32, 32, 3], dataType: .float32)
let isTrainingPlaceholder = graph.placeholder(shape: [1], dataType: .bool)
```

Start building

```
var x = conv2D(inputPlaceholder, filters: 2, kernelSize: (7, 7))
x = batchNormalization(x)
x = graph.reLU(with: x, name: nil)
x = maxPooling2D(x, kernelSize: (3, 3), strideSize: (2, 2))
```

Building Layers

maxPooling2D

```
let descriptor = MPSGraphPooling2DOpDescriptor(
  kernelWidth: kernelSize.width,
  kernelHeight: kernelSize.height,
  strideInX: strideSize.width,
  strideInY: strideSize.height,
  // Padding style is taken from TensorFlow
  paddingStyle: padding,
  // Data layout is batch / height / width / channels
  dataLayout: .NHWC)!
return graph.maxPooling2D(withSourceTensor: input, descriptor: descriptor, name: nil)
```

Building Layers

conv2D

Describe convolution

```
let desc = MPSGraphConvolution2DOpDescriptor(
  strideInX: strideSize.width,
  strideInY: strideSize.height,
  dilationRateInX: 1,
  dilationRateInY: 1,
  groups: 1,
  paddingStyle: padding,
  dataLayout: .NHWC,
  // Weights order is height / width / input / output
  weightsLayout: .HWIO)!

let fanIn = sourceTensor.shape!.totalElements
// Use `output` calculated from kernel size and stride
let fanOut = sourceTensor.shape![0].intValue * output.width * output.height * filters
// Randomly initialize weights
let convWeights = graph.randomWeights(
  shape: [kernelSize.height, kernelSize.width, sourceTensor.shape![3], filters],
  fanIn: fanIn,
  fanOut: fanOut)
weightTensors += [convWeights]

return graph.convolution2D(sourceTensor, weights: convWeights, descriptor: desc, name: name)
```

Building Layers

conv2D

Initialize weights

```
let desc = MPSGraphConvolution2DOpDescriptor(
  strideInX: strideSize.width,
  strideInY: strideSize.height,
  dilationRateInX: 1,
  dilationRateInY: 1,
  groups: 1,
  paddingStyle: padding,
  dataLayout: .NHWC,
  // Weights order is height / width / input / output
  weightsLayout: .HWIO)!

let fanIn = sourceTensor.shape!.totalElements
// Use `output` calculated from kernel size and stride
let fanOut = sourceTensor.shape![0].intValue * output.width * output.height * filters
// Randomly initialize weights
let convWeights = graph.randomWeights(
  shape: [kernelSize.height, kernelSize.width, sourceTensor.shape![3], filters],
  fanIn: fanIn,
  fanOut: fanOut)
weightTensors += [convWeights]

return graph.convolution2D(sourceTensor, weights: convWeights, descriptor: desc, name: name)
```

Weight Initialization

- Tensor flow uses GlorotUniform
- Draws samples from a uniform distribution within [-limit, limit]
- limit = $\sqrt{\frac{6}{fan\_in + fan\_out}}$
  - fan_in is the number of input units in the weight tensor and fan_out is the number of output units

Weight Initialization

Helper functions

```
func getRandomData(numValues: Int, minimum: Float, maximum: Float) -> [Float] {
    return (1...numValues).map { _ in Float.random(in: minimum..<maximum) }
}

extension Sequence where Element == NSNumber {
  var totalElements: Int {
    self.reduce(1) { $0 * $1.intValue }
  }
```

Number of elements from a Tensor’s shape

Random uniform distribution

Weight Initialization

Helper functions

```
extension Array where Element == Float {
  var data: Data {
    self.withUnsafeBytes { pointer in
      Data(bytes: pointer.baseAddress!, count: count * 4)
    }
  }

  func variableTensor(graph: MPSGraph, shape: [NSNumber]) -> MPSGraphTensor {
    assert(shape.totalElements == count)

    return graph.variable(with: data, shape: shape, dataType: .float32, name: nil)
  }
}
```

Variable tensor

Weight Initialization

Helper functions

```
extension MPSGraph {
  func uniformWeights(shape: [NSNumber], value: Float) -> MPSGraphTensor {
    return [Float](repeating: value, count: shape.totalElements)
      .variableTensor(graph: self, shape: shape)
  }

  func randomWeights(shape: [NSNumber], fanIn: Int, fanOut: Int) -> MPSGraphTensor {
    // Calculate GlorotUniform limit
    let limit = sqrt(6/Float(fanIn + fanOut))
    return getRandomData(numValues: shape.totalElements, minimum: -limit, maximum: limit)
      .variableTensor(graph: self, shape: shape)
  }
}
```

Building Layers

conv2D

Initialize weights

```
let desc = MPSGraphConvolution2DOpDescriptor(
  strideInX: strideSize.width,
  strideInY: strideSize.height,
  dilationRateInX: 1,
  dilationRateInY: 1,
  groups: 1,
  paddingStyle: padding,
  dataLayout: .NHWC,
  // Weights order is height / width / input / output
  weightsLayout: .HWIO)!

let fanIn = sourceTensor.shape!.totalElements
// Use `output` calculated from kernel size and stride
let fanOut = sourceTensor.shape![0].intValue * output.width * output.height * filters
// Randomly initialize weights
let convWeights = graph.randomWeights(
  shape: [kernelSize.height, kernelSize.width, sourceTensor.shape![3], filters],
  fanIn: fanIn,
  fanOut: fanOut)
weightTensors += [convWeights]

return graph.convolution2D(sourceTensor, weights: convWeights, descriptor: desc, name: name)
```

Building Layers

batchNormalization (initialize)

Initialize weights

```
let outputFeatures = input.shape![3].intValue
let beta = graph.uniformWeights(shape: [outputFeatures as NSNumber], value: 0)
let gamma = graph.uniformWeights(shape: [outputFeatures as NSNumber], value: 1)
weightTensors += [beta, gamma]

let shape = [1, 1, 1, NSNumber(value: outputFeatures)]
let movingMean = graph.uniformWeights(shape: shape, value: 0)
let movingVar = graph.uniformWeights(shape: shape, value: 1)
```

Building Layers

batchNormalization (initialize)

Initialize non-learnable params

```
let outputFeatures = input.shape![3].intValue
let beta = graph.uniformWeights(shape: [outputFeatures as NSNumber], value: 0)
let gamma = graph.uniformWeights(shape: [outputFeatures as NSNumber], value: 1)
weightTensors += [beta, gamma]

let shape = [1, 1, 1, NSNumber(value: outputFeatures)]
let movingMean = graph.uniformWeights(shape: shape, value: 0)
let movingVar = graph.uniformWeights(shape: shape, value: 1)
```

Building Layers
batchNormalization (train)

Control flow to compute mean and variance only when training

```
return graph.if(isTrainingPlaceholder, then: {
  // Graph for training
  return [...]
}, else: {
  // Graph for inference
  return [...]
}, name: nil)[0]
```

- Length of the returned arrays must be the same
- Sizes of corresponding tensors must be the same

Building Layers
batchNormalization (train)

Control dependency to have normalization dependent on updating params

```
// Calculate mean and variance
let axes: [NSNumber] = [0, 1, 2]
let mean = graph.mean(of: input, axes: axes, name: nil)
let variance = graph.variance(of: input, mean: mean, axes: axes, name: nil)
// Update parameters
let meanAssign = Self.addMovingVariable(graph, newValue: mean, existingValue: movingMean)
let varAssign = Self.addMovingVariable(graph, newValue: variance, existingValue: movingVar)
// Perform normalization
return self.graph.controlDependency(with: [meanAssign, varAssign], dependentBlock: {
  return [self.graph.normalize(input, mean: mean, variance: variance, gamma: gamma, beta:
    beta, epsilon: 0.001, name: nil)]
}, name: nil)
```

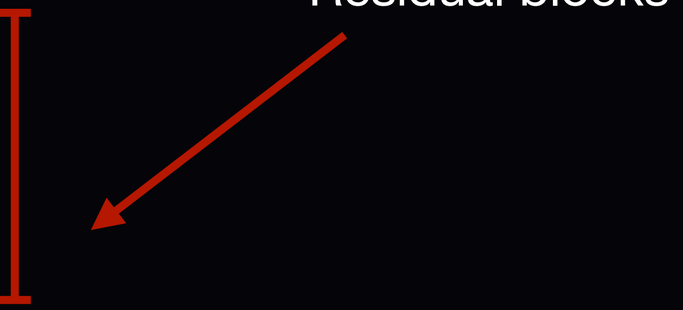
Identity Block

- Special case residual block with input/output same size

```
private func identityBlock(...) -> MPSGraphTensor {
  let (f1, f2) = filters
  var x = sourceTensor
  x = conv2D(x, filters: f1, kernelSize: (1, 1), strideSize: (1, 1), padding: .TF_VALID)
  x = batchNormalization(x)
  x = graph.reLU(with: x, name: nil)

  x = conv2D(x, filters: f1, kernelSize: (3, 3), strideSize: (1, 1), padding: .TF_SAME)
  x = batchNormalization(x)
  x = graph.reLU(with: x, name: nil)

  x = conv2D(x, filters: f2, kernelSize: (1, 1), strideSize: (1, 1), padding: .TF_VALID)
  x = batchNormalization(x)

  x = graph.addition(x, sourceTensor, name: nil)
  x = graph.reLU(with: x, name: nil)
  return x
}
```

Putting it all together

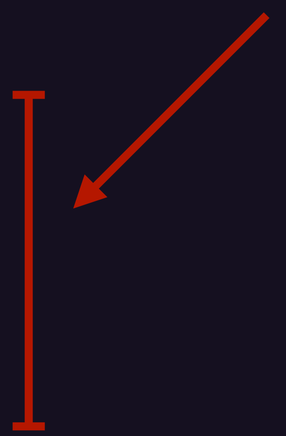
```
let inputPlaceholder = graph.placeholder(shape: [64, 32, 32, 3], dataType: .float32)
let outputPlaceholder = graph.placeholder(shape: [64, 10], dataType: .float32)
var x = conv2D(inputPlaceholder, filters: 2, kernelSize: (7, 7))
x = batchNormalization(x)
x = graph.reLU(with: x, name: nil)
x = maxPooling2D(x, kernelSize: (3, 3), strideSize: (2, 2))

x = convBlock(x, stride: 1, filters: (64, 256))
x = identityBlock(x, stride: 1, filters: (64, 256))
// Additional residual blocks
x = identityBlock(x, stride: 1, filters: (512, 2048))

x = avgPooling2D(x, kernelSize: (2, 2), padding: .TF_SAME)
let dataSize = x.shape![1...].totalElements
x = graph.reshape(x, shape: [64, dataSize as NSNumber], name: nil)
x = fullyConnected(x, outputFeatures: 10)
output = graph.softMax(with: x, axis: -1, name: nil)
```



Training the network

Gradients

Define loss



```
x = fullyConnected(x, outputFeatures: 10)
var loss = graph.softMaxCrossEntropy(
  x,
  labels: outputPlaceholder,
  axis: -1,
  reuctionType: .sum,
  name: nil)
loss = graph.division(loss, graph.constant(Double(batchSize), dataType: .float32), name: nil)
// Compute partial derivatives for each weight
let gradients = graph.gradients(of: loss, with: weightTensors, name: nil)
let learningRate = graph.constant(0.01, dataType: .float32)
var updateOps = [MPSGraphOperation]()
// Update each weight
for (key, value) in gradients {
  let newValue = graph.stochasticGradientDescent(learningRate: learningRate, values: key,
gradient: value, name: nil)
  updateOps.append(graph.assign(key, tensor: newValue, name: nil))
}
```

Training the network

Gradients

Create gradients

```
x = fullyConnected(x, outputFeatures: 10)
var loss = graph.softMaxCrossEntropy(
  x,
  labels: outputPlaceholder,
  axis: -1,
  reuctionType: .sum,
  name: nil)
loss = graph.division(loss, graph.constant(Double(batchSize), dataType: .float32), name: nil)
// Compute partial derivatives for each weight
let gradients = graph.gradients(of: loss, with: weightTensors, name: nil)
let learningRate = graph.constant(0.01, dataType: .float32)
var updateOps = [MPSGraphOperation]()
// Update each weight
for (key, value) in gradients {
  let newValue = graph.stochasticGradientDescent(learningRate: learningRate, values: key,
gradient: value, name: nil)
  updateOps.append(graph.assign(key, tensor: newValue, name: nil))
}
```

This is why we kept a reference to all the weight tensors

Training the network

Gradients

```
x = fullyConnected(x, outputFeatures: 10)
var loss = graph.softMaxCrossEntropy(
  x,
  labels: outputPlaceholder,
  axis: -1,
  reuctionType: .sum,
  name: nil)
loss = graph.division(loss, graph.constant(Double(batchSize), dataType: .float32), name: nil)
// Compute partial derivatives for each weight
let gradients = graph.gradients(of: loss, with: weightTensors, name: nil)
let learningRate = graph.constant(0.01, dataType: .float32)
var updateOps = [MPSGraphOperation]()
// Update each weight
for (key, value) in gradients {
  let newValue = graph.stochasticGradientDescent(learningRate: learningRate, values: key,
gradient: value, name: nil)
  updateOps.append(graph.assign(key, tensor: newValue, name: nil))
}
```

Use gradients to update weights

Learning rate is how quickly we move in the direction of the gradient

Training the network

Provide input data

```
let inputFloats: [Float] = loadInput()
let inputData = MPSNDArray(device: gDevice, descriptor: MPSNDArrayDescriptor(dataType:
  .float32, shape: [64, 32, 32, 3]))
inputData.readBytes(&inputFloats, strideBytes: nil)
let batchInput = MPSGraphTensor(inputData)
let batchLabels = ... // Tensor for labels same as above

// Indicate we are training
let isTraining = MPSGraphTensorData(
  device: MPSGraphDevice(mtlDevice: gDevice),
  data: Data(bytes: [1], count: 1),
  shape: [1],
  dataType: .bool)
// Run graph with inputs and targets
graph.runAsync(
  feeds: [
    inputPlaceholder: batch,
    isTrainingPlaceholder: isTraining,
    outputPlaceholder: batchLabels,
  ],
  targetTensors: [loss, output],
  targetOperations: updateOps,
  executionDescriptor: desc)
```

Training the network

Handle results

```
let desc = MPSGraphExecutionDescriptor()
desc.completionHandler = { results, error in
  var lossFloat: Float = 0
  results[loss]!.mpsndarray().readBytes(&lossFloat, strideBytes: nil)

  var outputFloat = [Float](repeating: 0, count: 10*64)
  results[output]!.mpsndarray().readBytes(&outputFloat, strideBytes: nil)
}
```

Demo Time

Tips

- Conditional performance penalty
- Validate size of outputs
- Use header comments (but not always correct)

```
/// Defines a scope where all the ops defined in this block get controlDependency operations
///
/// - Returns: A valid MPSGraphTensor array with results forwarded to return of controlDependency call

/// Runs the graph for given feeds to return targetTensor values, ensuring all target operations also executed. This call blocks till execution has completed.
///
/// - Parameters:
///   - operations: Operations maked as control dependency for all ops created inside the dependent block
///   - dependentBlock: MPSGraphControlFlowDependencyBlock which is provided by caller to create dependent ops
///   - name: name of scope
/// - Returns: A valid MPSGraphTensor array with results returned from dependentBlock forwarded
open func controlDependency(with operations: [MPSGraphOperation], dependentBlock: @escaping MPSGraphControlFlowDependencyBlock, name:
    String?) -> [MPSGraphTensor]
```

Conclusion

- https://github.com/EmergeTools/talks
- Find me at the Emerge Tools booth to chat about app performance!
- Get in touch @sond813