ML Summit '20

introduction to swift for tensorflow

brett koonce
cto, quarkworks
convolutionalneuralnetworkswithswift.com

#MLSummit

overview

- goal: understand what is happening when swift for tensorflow is running mnist demo
- s4tf: swift, llvm, neural networks, autodiff, xla
- demos of various hardware: cpu, gpu, tpu
- recap, next steps

swift for tensorflow: components

- swift programming language, packages
- llvm compiler
- swift-api neural network operators
- autodiff
- xla ⟶ tensorflow + hardware

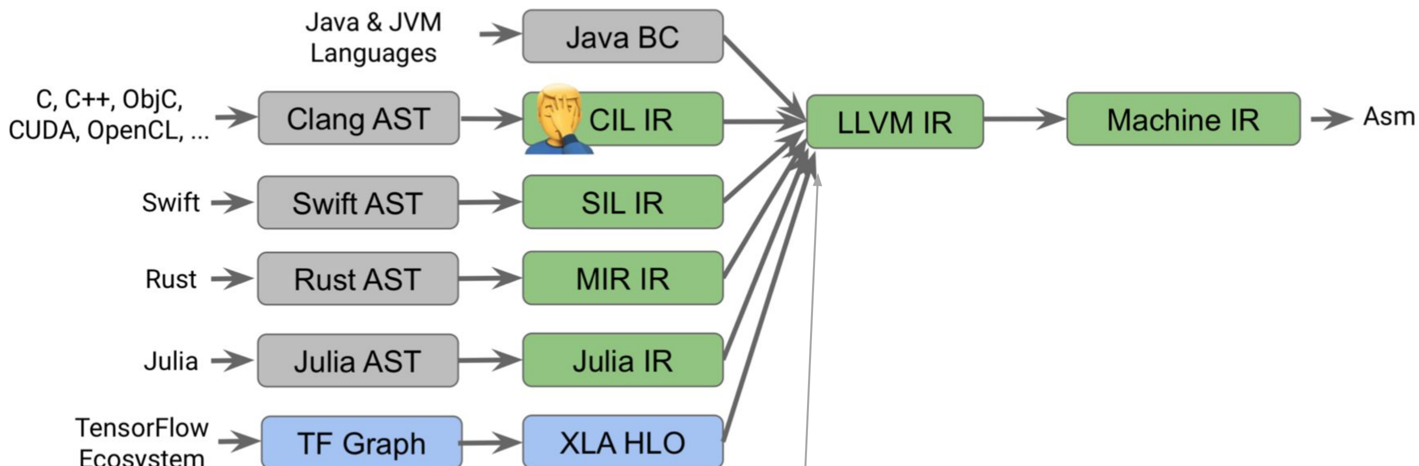
before swift

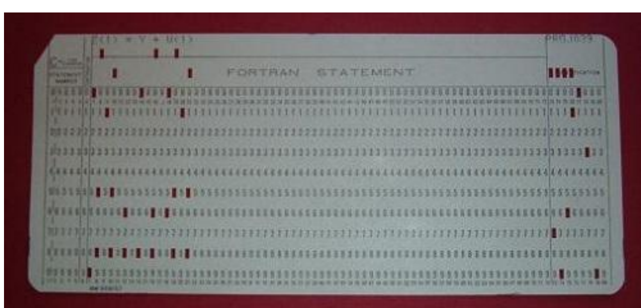
- assembly → c
- smalltalk → objective-c
- speed vs. safety
- embedded/edge resources

swift

- legacy interoperability
- open source (2015), cross platform
- functional programming
- type safety

$\Sigma^* \longrightarrow$ ir $\longrightarrow$ llvm



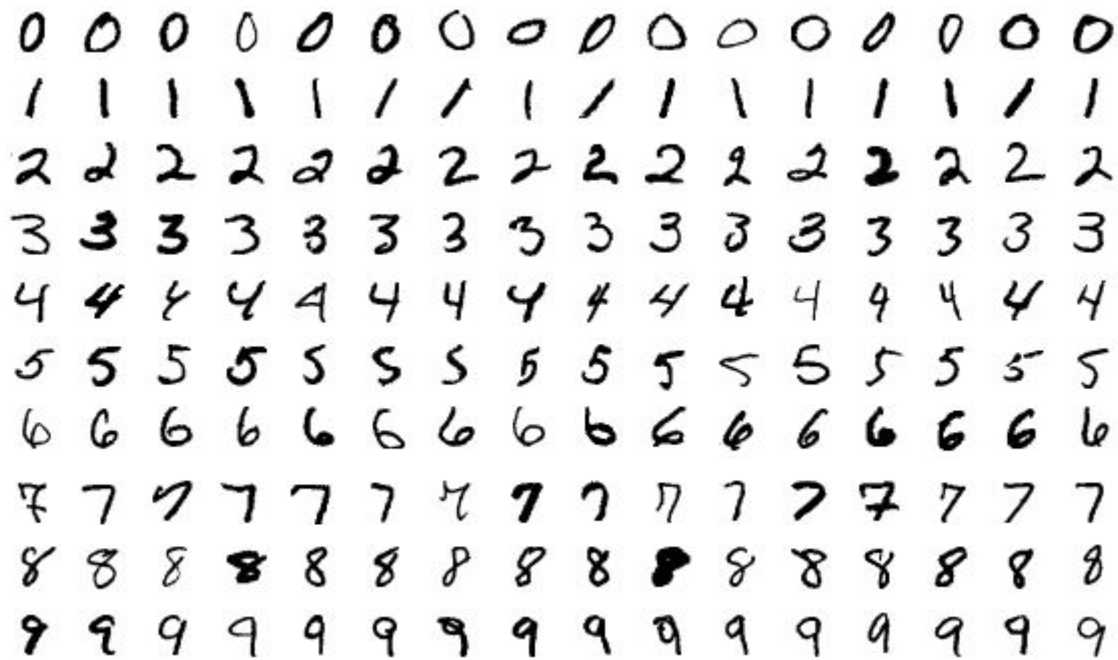


flang $\longrightarrow$ fir

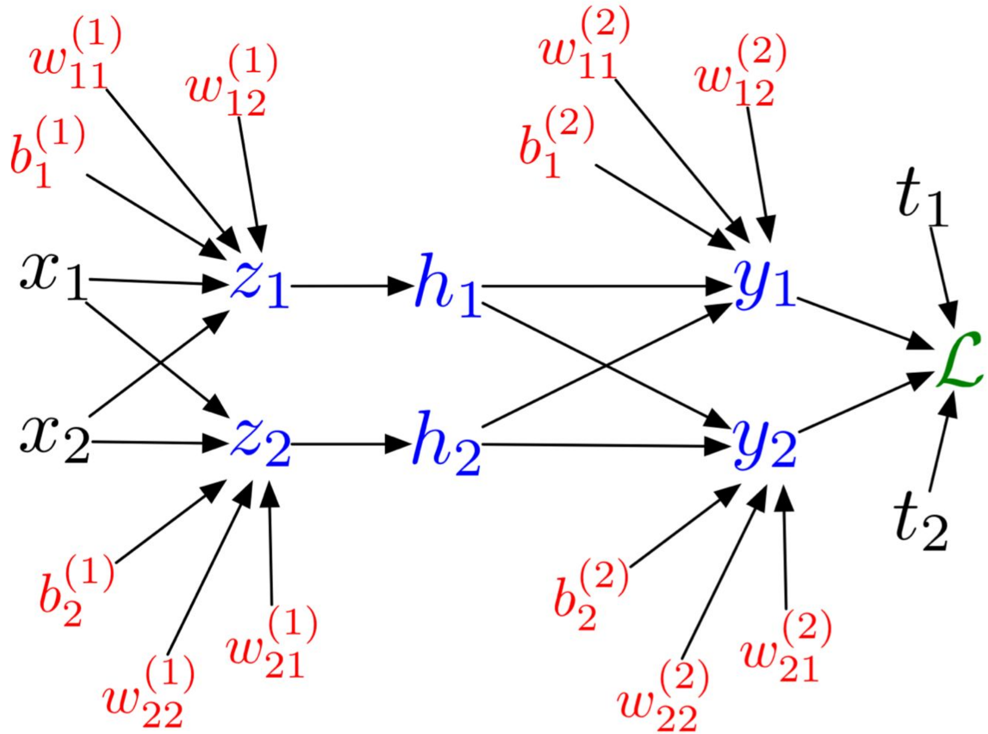
swift-models + swift package manager

```swift
dependencies: [
  .package(
    name: "swift-models", url: "https://github.com/tensorflow/swift-models.git",
        .branch("master")
  ),
],
```

```swift
import Datasets
import TensorFlow
```

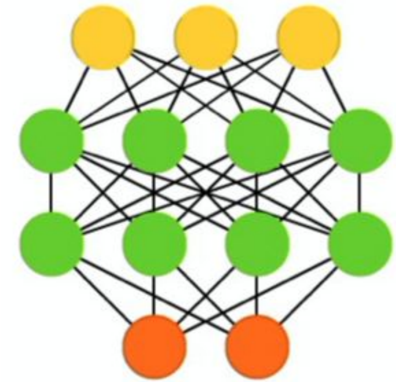


multi-layer perceptron

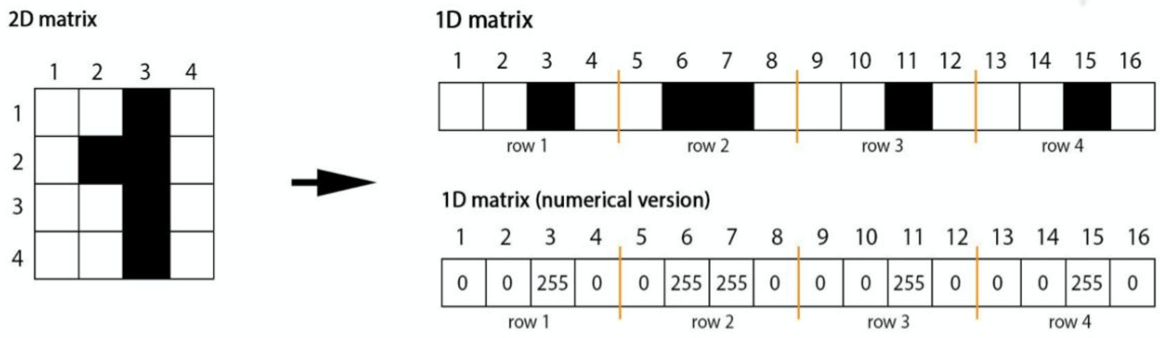


swift mlp

```swift
struct MLP: Layer {
  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 784, outputSize: 512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize: 512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return input.sequenced(through: flatten, inputLayer, hiddenLayer, outputLayer)
  }
}
```



2D matrix to 1D matrix conversion

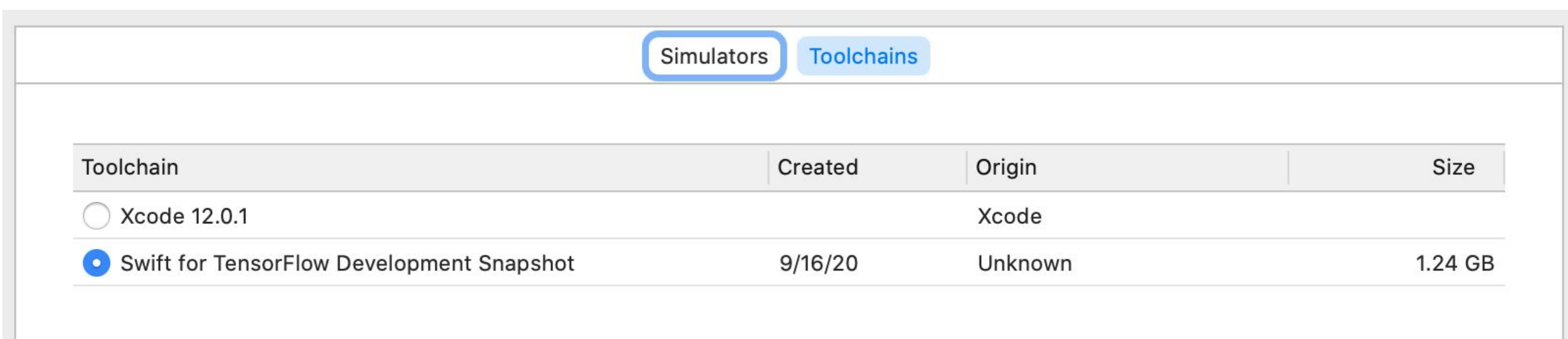


layer protocol

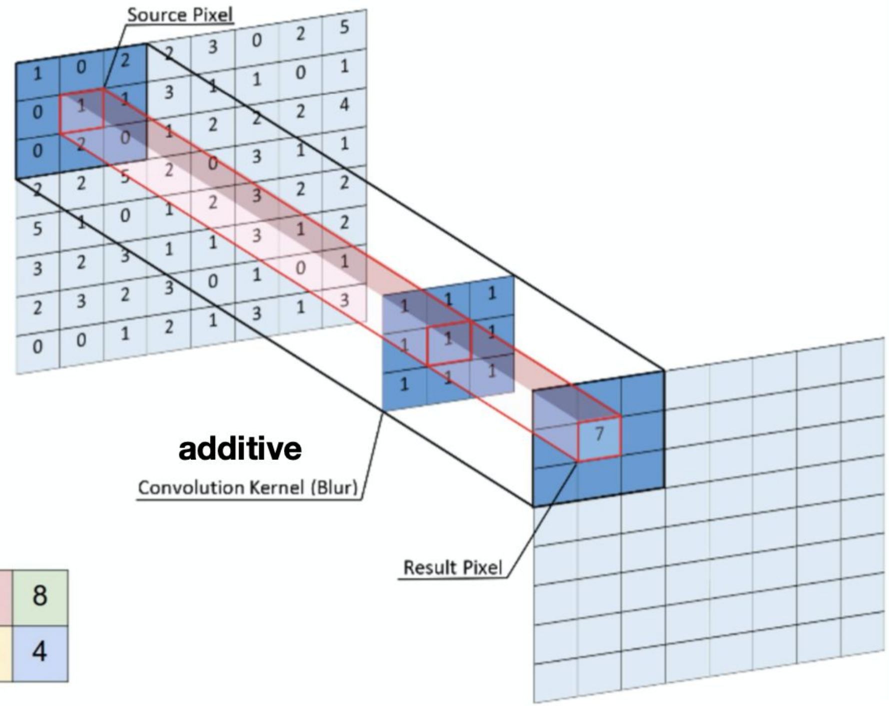
```swift
/// A neural network layer.
///
/// Types that conform to `Layer` represent functions that map inputs to outputs. They may have an
/// internal state represented by parameters, such as weight tensors.
///
/// `Layer` instances define a differentiable `callAsFunction(_:)` method for mapping inputs to
/// outputs.
public protocol Layer: Module where Input: Differentiable {
  /// Returns the output obtained from applying the layer to the given input.
  ///
  /// - Parameter input: The input to the layer.
  /// - Returns: The output.
  @differentiable
  func callAsFunction(_ input: Input) -> Output

  @differentiable
  func forward(_ input: Input) -> Output
}
```

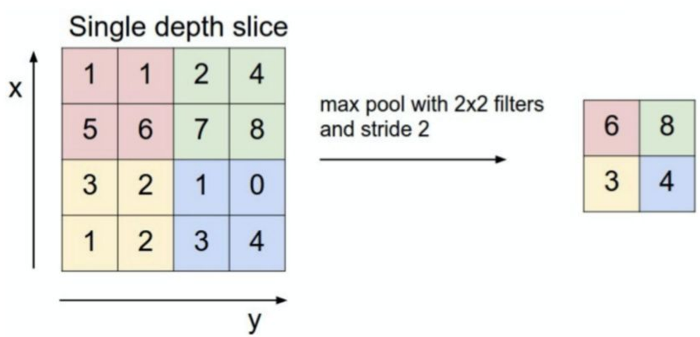
1d mnist demo

- xcode + s4tf toolchain



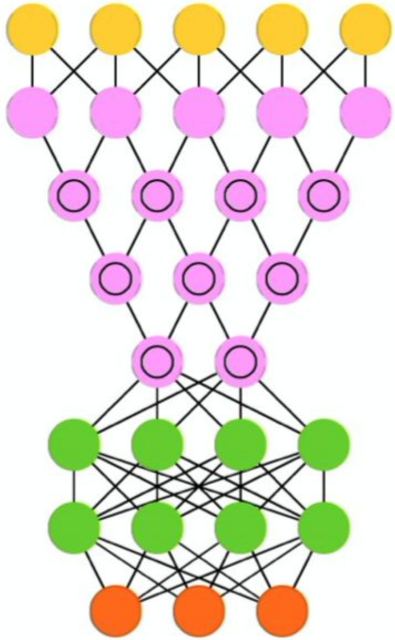
convolutions





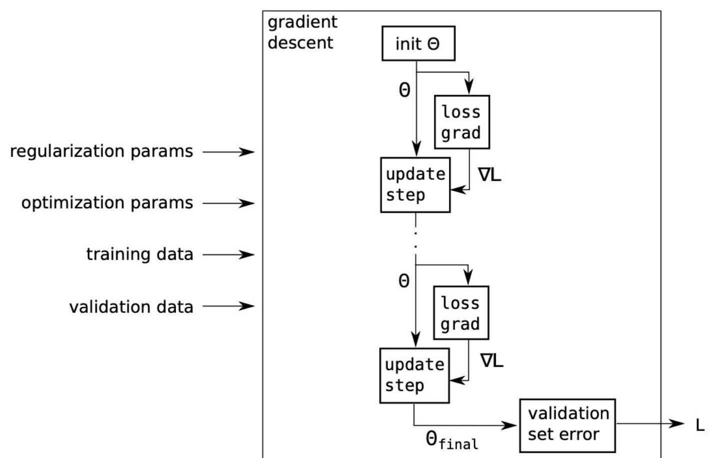
convolutional neural network

```swift
struct CNN: Layer {
  var conv1a = Conv2D<Float>(filterShape: (3, 3, 1, 32), padding: .same, activation: relu)
  var conv1b = Conv2D<Float>(filterShape: (3, 3, 32, 32), padding: .same, activation: relu)
  var pool1 = MaxPool2D<Float>(poolSize: (2, 2), strides: (2, 2))

  var flatten = Flatten<Float>()
  var inputLayer = Dense<Float>(inputSize: 14 * 14 * 32, outputSize: 512, activation: relu)
  var hiddenLayer = Dense<Float>(inputSize: 512, outputSize: 512, activation: relu)
  var outputLayer = Dense<Float>(inputSize: 512, outputSize: 10)

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolutionLayer = input.sequenced(through: conv1a, conv1b, pool1)
    return convolutionLayer.sequenced(through: flatten, inputLayer, hiddenLayer, outputLayer)
  }
}
```



sgd training loop

```swift
let batchSize = 128
let epochCount = 12
var model = CNN()
let optimizer = SGD(for: model, learningRate: 0.1)
let dataset = MNIST(batchSize: batchSize)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.prefix(epochCount).enumerated() {
  // SGD goes here
```



gradient update step

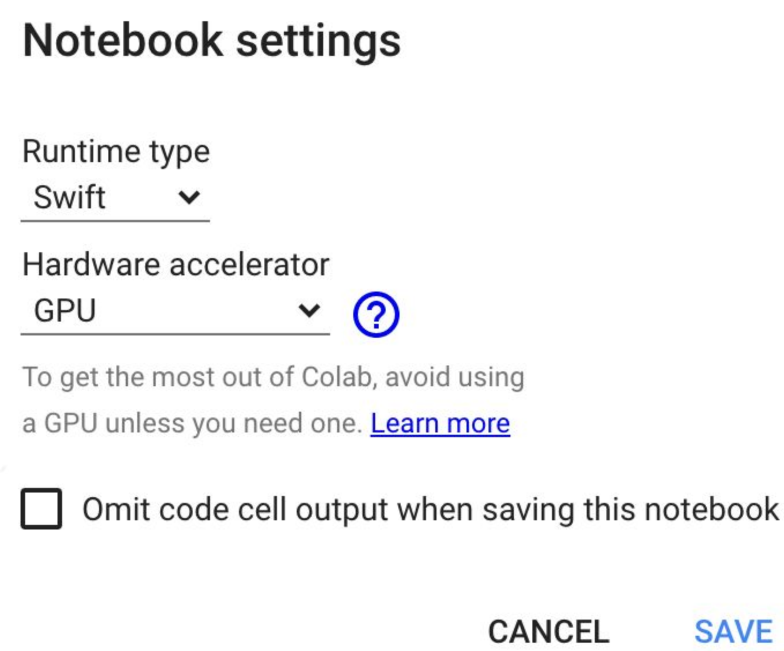
```
Context.local.learningPhase = .training
for batch in epochBatches {
  let (images, labels) = (batch.data, batch.label)
  let (_, gradients) = valueWithGradient(at: model) { model -> Tensor<Float> in
    let logits = model(images)
    return softmaxCrossEntropy(logits: logits, labels: labels)
  }
  optimizer.update(&model, along: gradients)
}
```

validation

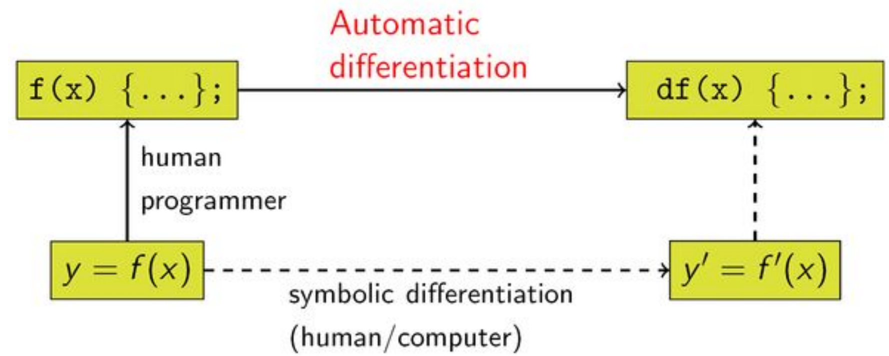
```
Context.local.learningPhase = .inference
var testLossSum: Float = 0
var testBatchCount = 0
var correctGuessCount = 0
var totalGuessCount = 0
for batch in dataset.validation {
  let (images, labels) = (batch.data, batch.label)
  let logits = model(images)
  testLossSum += softmaxCrossEntropy(logits: logits, labels: labels).scalarized()
  testBatchCount += 1

  let correctPredictions = logits.argmax(squeezingAxis: 1) .== labels
  correctGuessCount += Int(Tensor<Int32>(correctPredictions).sum().scalarized())
  totalGuessCount = totalGuessCount + batch.data.shape[0]
}

let accuracy = Float(correctGuessCount) / Float(totalGuessCount)
print(
  """
  [Epoch \(epoch + 1)] \
  Accuracy: \(correctGuessCount)/\(totalGuessCount) (\(accuracy)) \
  Loss: \(testLossSum / Float(testBatchCount))
  """
)
```

2d mnist demo

- colab swift kernel + gpu in cloud



autodiff



- history
- symbolic approaches
- absolutely vs. approximately correct
    - Mathematica's derivatives for one layer of soft ReLU (univariate case):

```
D[Log[1 + Exp[w * x + b]], w]
```

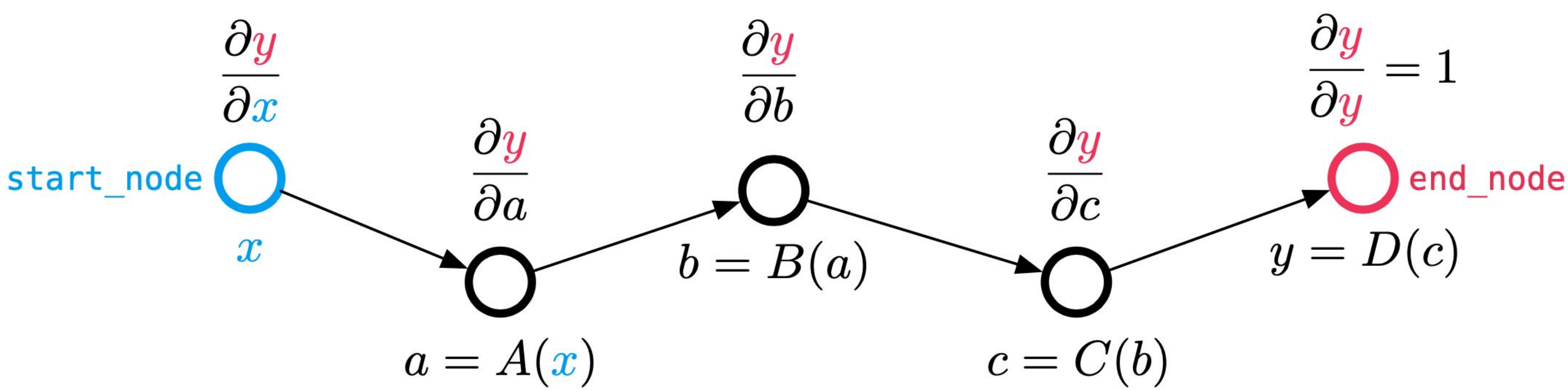
Out[11]= $$\frac{e^{b+w\,x}\,w}{1+e^{b+w\,x}}$$

    - Derivatives for two layers of soft ReLU:

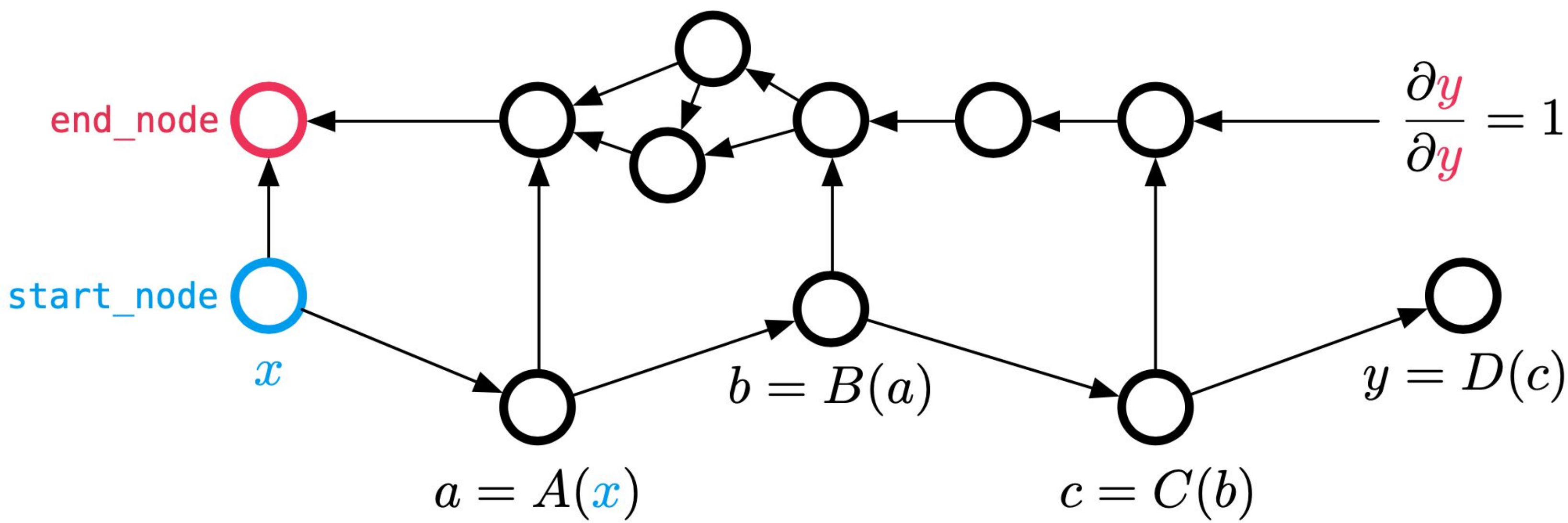
```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
```

Out[19]= $$\frac{e^{b1+b2+w1\,x+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\,w2\,x}{\left(1+e^{b1+w1\,x}\right)\left(1+e^{b2+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\right)}$$

chain rule

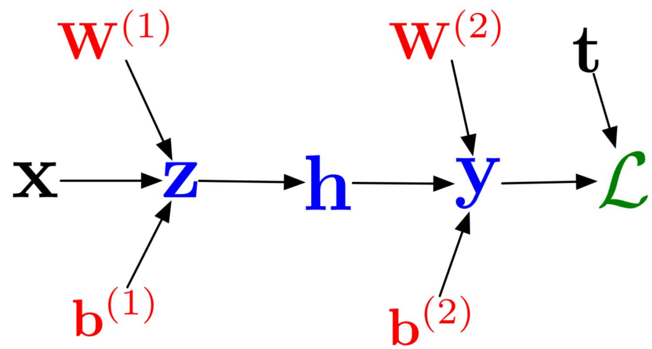

$\frac{\partial y}{\partial x}$
$\frac{\partial y}{\partial a}$
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$
start_node
end_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$

backprop



end_node
start_node
x
$\frac{\partial y}{\partial y} = 1$
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$

vector jacobian product



The backprop equation (single child node) can be written as a vector-Jacobian product (VJP):

$$\overline{x_j} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_j} \qquad \qquad \overline{\mathbf{x}} = \overline{\mathbf{y}}^\top \mathbf{J}$$

That gives a row vector. We can treat it as a column vector by taking

$$\overline{\mathbf{x}} = \mathbf{J}^\top \overline{\mathbf{y}}$$

swish: naive

```
/// Returns a tensor by applying the swish activation function, namely
/// `x * sigmoid(x)`.
///
/// Source: "Searching for Activation Functions" (Ramachandran et al. 2017)
/// https://arxiv.org/abs/1710.05941
@inlinable
@differentiable
public func swish<T: TensorFlowFloatingPoint>(_ x: Tensor<T>) -> Tensor<T> {
  x * sigmoid(x)
}
```

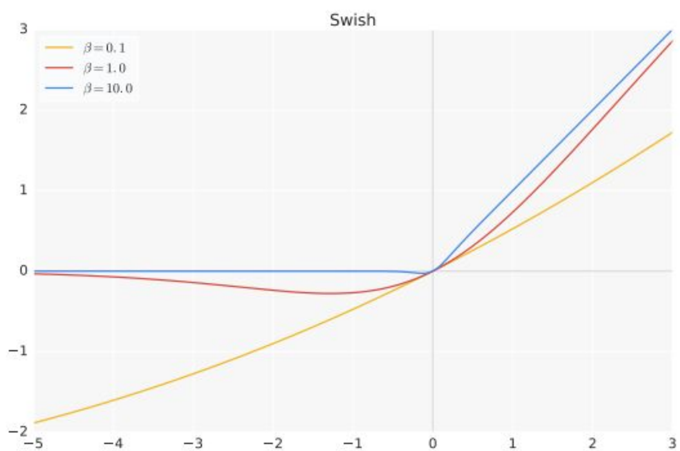


Figure 4: The Swish activation function.

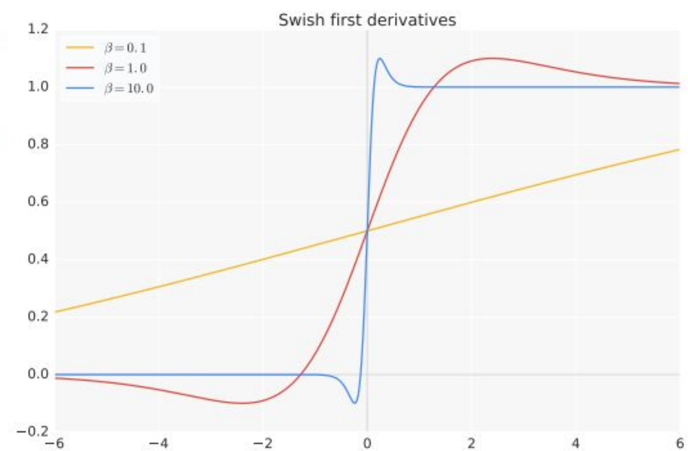


Figure 5: First derivatives of Swish.

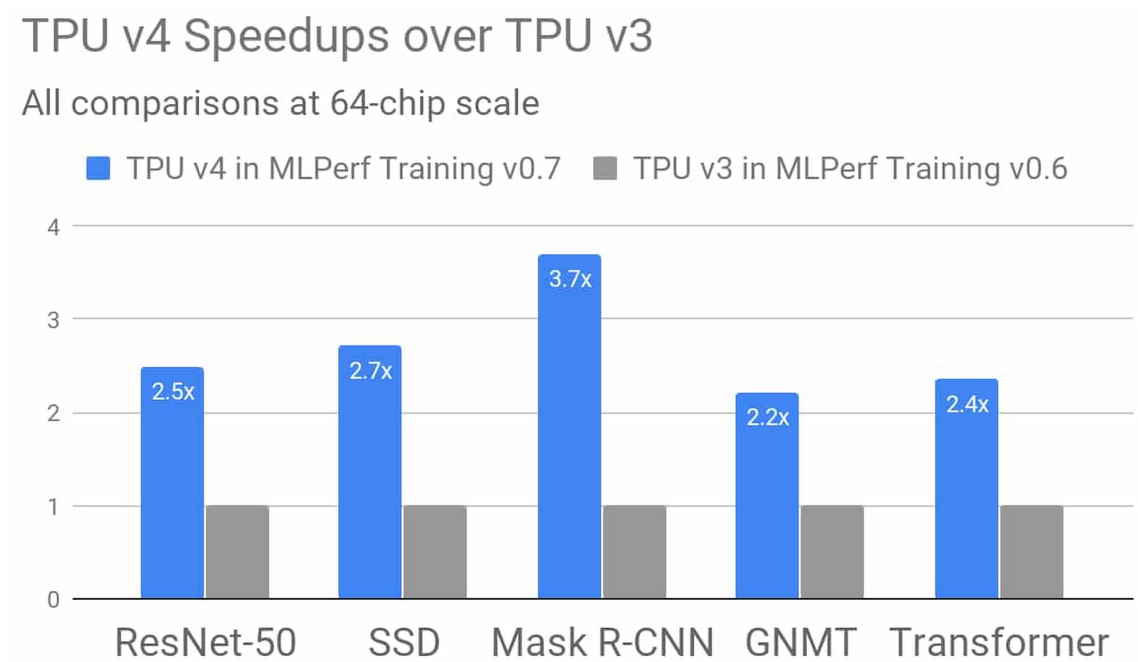
swish: custom vjp

```swift
// Note: A custom vjp function for swish is required to avoid excessive
// tensor memory consumption due to storing both `x` and `sigmoid(x)` for
// backprop. This vjp recomputes `sigmoid(x)` during backprop, so that
// the `sigmoid(x)` expression can be freed during the forward pass.
@inlinable
@derivative(of: swish)
func _vjpSwish<T: TensorFlowFloatingPoint>(
  _ x: Tensor<T>
) -> (value: Tensor<T>, pullback: (Tensor<T>) -> Tensor<T>) {
  return (
    swish(x),
    { v in
      let sigmoidFeatures = sigmoid(x)
      let grad = sigmoidFeatures * (1.0 + x * (1 - sigmoidFeatures))
      return grad * v
    }
  )
}
```

tpu-v3-2048: >100 petaflop, 32tb ram

xla

- tensorflow
- jax
- julia
- pytorch
- s4tf



mnist + xla + tpu

```
let device = Device.defaultXLA
model.move(to: device)
optimizer = SGD(copying: optimizer, to: device)

print("Starting training...")

for (epoch, epochBatches) in dataset.training.prefix(epochCount).enumerated() {
  Context.local.learningPhase = .training
  for batch in epochBatches {
    let (images, labels) = (batch.data, batch.label)
    let deviceImages = Tensor(copying: images, to: device)
    let deviceLabels = Tensor(copying: labels, to: device)
    let (_, gradients) = valueWithGradient(at: model) { model -> Tensor<Float> in
      let logits = model(deviceImages)
      return softmaxCrossEntropy(logits: logits, labels: deviceLabels)
    }
    optimizer.update(&model, along: gradients)
    LazyTensorBarrier()
  }
```

2d mnist xla + tpu demo

- gcloud compute tpus create s4tf-mnist-demo --zone=us-central1-f --accelerator-type=v2-8 --version=nightly
- shell vars:

```
export PATH=~/usr/bin:"${PATH}"
export TPU_IP_ADDRESS=10.9.170.90

export XLA_USE_XRT=1
export XRT_TPU_CONFIG="tpu_worker;0;$TPU_IP_ADDRESS:8470"
export XRT_WORKERS='localservice:0;grpc://localhost:40934'
export XRT_DEVICE_MAP="TPU:0;/job:localservice/replica:0/task:0/device:TPU:0"
```

recap

- we built and trained a simple convolutional neural network
- swift, llvm, neural networks, autodiff, xla
- ran it locally and in the cloud using cpu, gpu, tpu

next steps

- swift-models + colab
- 📓: cnn's w/ s4tf
- swift-sig meetings: 9am pst fridays (thanks ewa!)
- autodiff: roger grosse
- cloud tpu tutorials, tfrc