Sample Code

Training a neural network to recognize digits

Build a simple neural network and train it to recognize randomly generated numbers.

Download

macOS 12.0+ | Xcode 14.3+

Overview

This sample code project uses the Basic Neural Network Subroutines (BNNS) library to create a simple neural network that's capable of recognizing digits.

The sample iterates over randomly generated digits in the training phase, incrementally improving its ability to recognize numbers. After the code completes the training phase, it evaluates its accuracy at recognizing numbers, and returns a score.

A 6 x 6 matrix represents each digit. For example, the code below represents the number 3:

```
static let three: [Float] = [0, 1, 1, 1, 1, 0,  // ○●●●●○
                             0, 0, 0, 0, 1, 0,  // ○○○○●○
                             0, 1, 1, 1, 1, 0,  // ○●●●●○
                             0, 0, 0, 0, 1, 0,  // ○○○○●○
                             0, 1, 1, 1, 1, 0,  // ○●●●●○
                             0, 0, 0, 0, 0, 0]  // ○○○○○○
```

The network consists of the three layers below:

- Fused convolution-batch normalization layer

- Pooling layer
- Fully connected layer

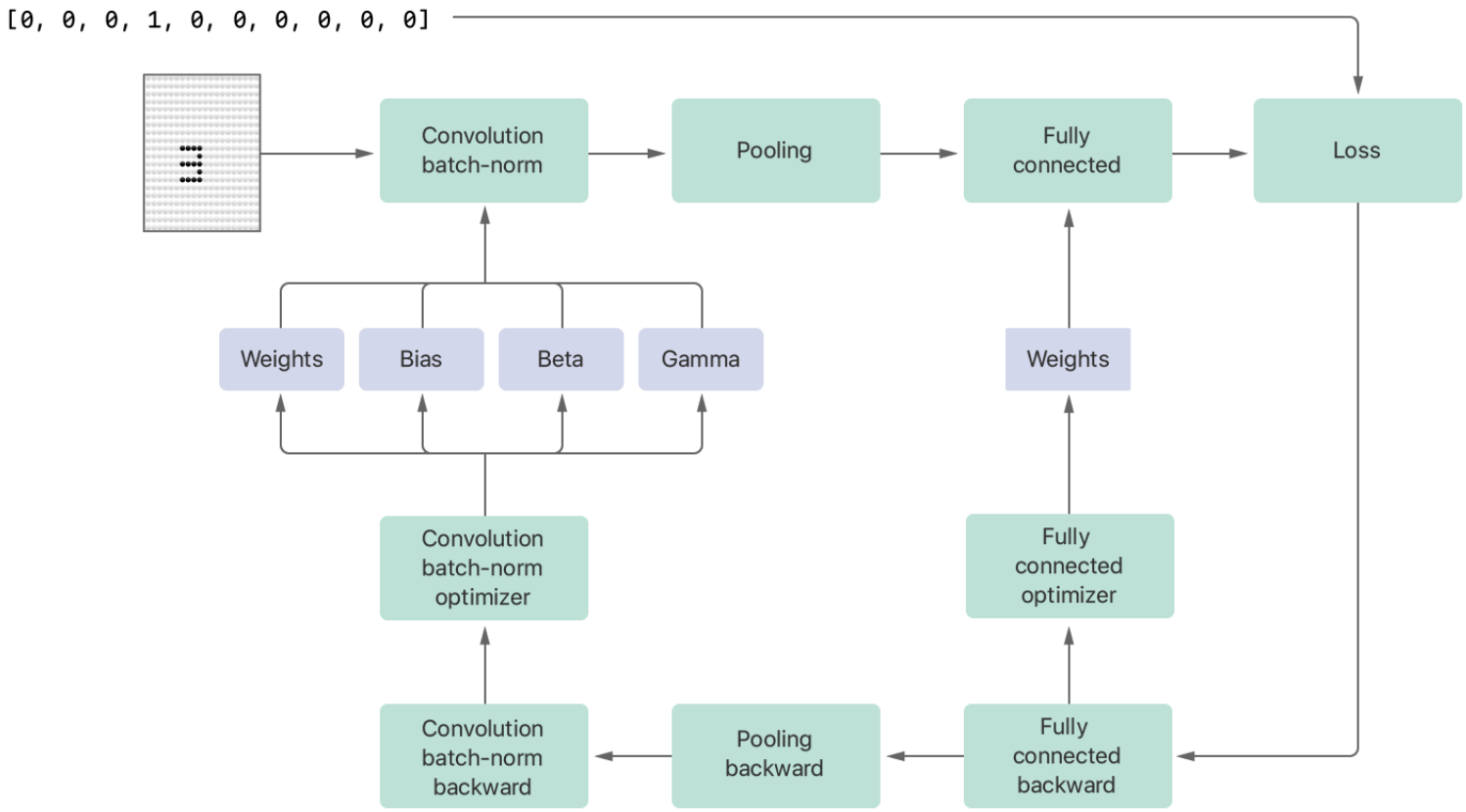
After the code completes a forward pass, it calculates its loss, which is a score that indicates how the predicted values deviate from the labels. The sample code project uses the gradients that the loss generates as the basis for the backward pass, where it backward-applies the three layers in reverse order.

The backward passes generate gradient values that an optimizer uses to update the parameters below:

- Convolution weights that the app initializes with random values, and bias
- Normalization beta (offset) and gamma (scale)
- Fully connected weights

The optimizer's gradual changes to the weights, bias, beta, and gamma increases the network's efficacy in recognizing digits with each iteration.

The image below shows the relationships between the layers:



Define the filter parameters

The sample creates a `BNNSFilterParameters` structure with `useClientPointer`. This flag instructs the layers to keep the provided pointers at creation time, and to work directly from that data rather than use internal copies of the data.

```
static var filterParameters = BNNSFilterParameters(
    flags: BNNSFlags.useClientPointer.rawValue,
    n_threads: 1,
    alloc_memory: nil,
    free_memory: nil)
```

Create the fused convolution-batch normalization layer

The convolution-batch normalization layer consists of two sublayers.

- Convolution layers that generate their output by multiplying each input value and its neighbors by corresponding values in an array of weights, and then adding a corresponding bias. Each output value is the sum of each of those operations. Convolution layers are fundamental to convolutional neural networks and, with the correct weights and bias values, can identify features, such as horizontal and vertical lines.
- Normalization layers that rescale their data so that all of the batches of data have the same standard deviation.

The app initializes the convolution weights array with random values.

```
static let convolutionWeights: BNNSNDArrayDescriptor = {
    let convolutionKernelSize = 3

    let convolutionWeightsShape = BNNS.Shape.convolutionWeightsOIHW(
        convolutionKernelSize,
        convolutionKernelSize,
        convolutionInputImageChannels,
        convolutionOutputImageChannels)

    guard let desc = BNNSNDArrayDescriptor.allocate(
        randomUniformUsing: randomGenerator,
        range: Float(-0.5)...Float(0.5),
        shape: convolutionWeightsShape) else {
        fatalError("Unable to create `convolutionWeightsArray`.")
    }
    return desc
}()
```

The app initializes the convolution bias and the batch normalization beta and gamma arrays with a repeated scalar value.

```
static let convolutionBias = BNNSNDArrayDescriptor.allocate(
    repeating: Float(0),
    shape: .vector(convolutionOutputImageChannels))

static let featureMaps = convolutionOutputImageChannels

static let batchNormBeta = BNNSNDArrayDescriptor.allocate(
    repeating: Float(0),
    shape: .vector(featureMaps),
    batchSize: batchSize)

static let batchNormGamma = BNNSNDArrayDescriptor.allocate(
    repeating: Float(1),
    shape: .vector(featureMaps),
    batchSize: batchSize)
```

The code below creates the fused layer that applies convolution and normalization to the input:

```
static let fusedConvBatchNormLayer: BNNS.FusedParametersLayer = {

    let convolutionParameters = BNNS.FusedConvolutionParameters(
        type: .standard,
        weights: convolutionWeights,
        bias: convolutionBias,
        stride: (1, 1),
        dilationStride: (1, 1),
        groupSize: 1,
        padding: .symmetric(x: convolutionPadding,
                            y: convolutionPadding))

    let normalizationParameters = BNNS.FusedNormalizationParameters(
        type: .batch(movingMean: batchNormMovingMean,
                     movingVariance: batchNormMovingVariance),
        beta: batchNormBeta,
        gamma: batchNormGamma,
        momentum: 0.9,
        epsilon: 1e-07,
        activation: .rectifiedLinear)

    guard let layer = BNNS.FusedParametersLayer(
        input: input,
        output: batchNormOutput,
```

```
        fusedLayerParameters: [convolutionParameters, normalizationParameters],
        filterParameters: filterParameters) else {
        fatalError("unable to create fusedConvBatchnormLayer")
    }

    return layer
}()
```

Create the pooling layer

Pooling layers downscale their input while preserving the most important information and produce an output that, in the case of this sample code project, consists of the maximum value in each input pixel's local neighborhood.

The following code creates the pooling layer:

```
static var poolingLayer: BNNS.PoolingLayer = {
    guard let poolingLayer = BNNS.PoolingLayer(
        type: .max(xDilationStride: 1, yDilationStride: 1),
        input: batchNormOutput,
        output: poolingOutput,
        bias: nil,
        activation: .identity,
        kernelSize: (2, 2),
        stride: (2, 2),
        padding: .zero,
        filterParameters: filterParameters) else {
        fatalError("Unable to create `poolingLayer`.")
    }

    return poolingLayer
}()
```

Create the fully connected layer

Fully connected layers compute the matrix-vector product of a weights matrix and its input, and flatten the data to predict the correct label.

The app initializes the fully connected weights array with random values.

```
static let fullyConnectedWeights: BNNSNDArrayDescriptor = {
    guard let desc = BNNSNDArrayDescriptor.allocate(
        randomUniformUsing: randomGenerator,
        range: Float(-0.5)...Float(0.5),
        shape: .matrixRowMajor(poolingOutputSize,
                               fullyConnectedOutputWidth)) else {
        fatalError("Unable to create `fullyConnectedWeightsArray`.")
    }
    return desc
}()
```

The code below creates the fully connected layer:

```
static var fullyConnectedLayer: BNNS.FullyConnectedLayer = {

    let desc = BNNSNDArrayDescriptor(dataType: .float,
                                     shape: .vector(poolingOutputSize))

    guard let fullyConnectedLayer = BNNS.FullyConnectedLayer(
            input: desc,
            output: fullyConnectedOutput,
            weights: fullyConnectedWeights,
            bias: nil,
            activation: .identity,
            filterParameters: filterParameters) else {
        fatalError("Unable to create `fullyConnectedLayer`.")
    }

    return fullyConnectedLayer
}()
```

Create the loss layer

The loss layer is responsible for quantifying a score that indicates how the predicted values deviate from the labels.

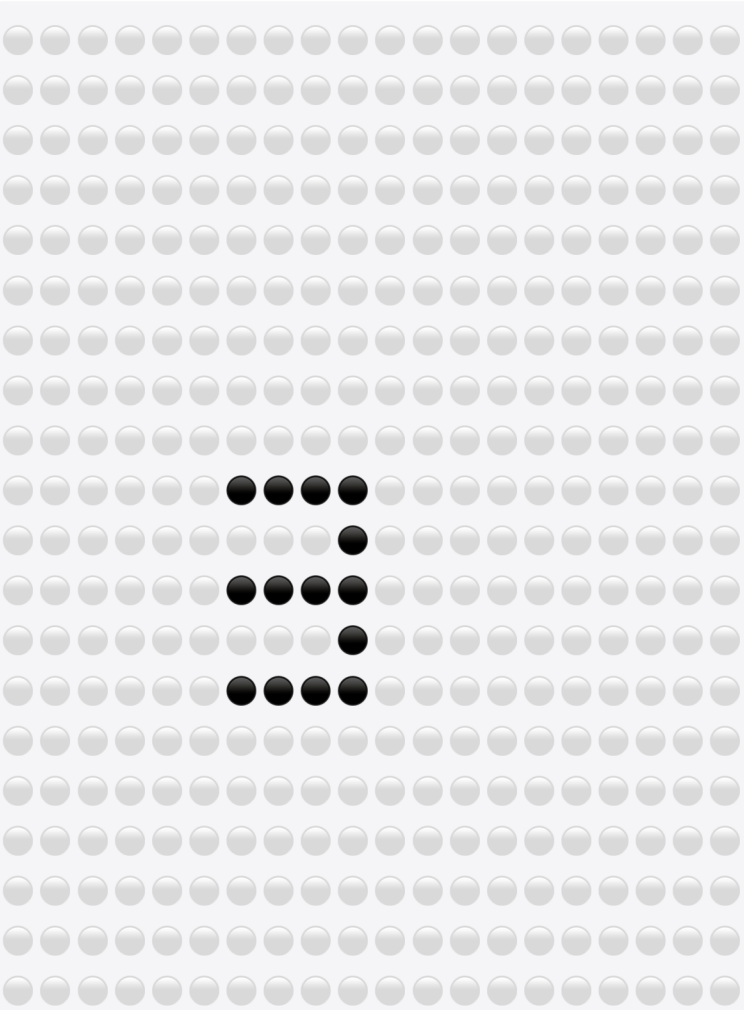
The code below creates the loss layer:

```
static var lossLayer: BNNS.LossLayer = {

    guard let lossLayer = BNNS.LossLayer(input: fullyConnectedOutput,
```

```
                                        output: lossOutput,
                                        lossFunction: .softmaxCrossEntropy(labelSmo
                                        lossReduction: .reductionMean,
                                        filterParameters: filterParameters) else {
        fatalError("Unable to create `lossLayer`.")
    }

    return lossLayer
}()
```

Create the candidate input

For each iteration of the training phase, the sample creates a matrix that represents a random digit, and a *one-hot* encoded tensor of the same digit. The sample places digits randomly in a 20 x 20 matrix, so a 3 might appear in the matrix as the image below. This example renders `0` as ⚪, and 1 as ⚫.



The one-hot encoded tensor contains a 1 at the zero-based index of 3.

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

The sample code project uses a batch size of 32, so each iteration generates 32 random digits in random positions in the 20 x 20 grid.

Perform the forward pass

To perform the forward pass, the sample code calls `apply` on the fused, pooling, and fully connected layers.

```
try fusedConvBatchNormLayer.apply(batchSize: batchSize,
                                  input: input,
                                  output: batchNormOutput,
                                  for: .training)

try poolingLayer.apply(batchSize: batchSize,
                       input: batchNormOutput,
                       output: poolingOutput)

try fullyConnectedLayer.apply(batchSize: batchSize,
                              input: poolingOutput,
                              output: fullyConnectedOutput)
```

Calculate the loss and loss gradient

Calculating the loss evaluates the efficacy of the neural network. The loss layer generates its output, `lossOutput`, which contains a score that indicates how the predicted values deviate from the labels, and `lossInputGradient`, which is the output gradient parameter to the backward application of the fully connected layer.

```
try lossLayer.apply(batchSize: batchSize,
                    input: fullyConnectedOutput,
                    labels: oneHotLabels,
                    output: lossOutput,
                    generatingInputGradient: lossInputGradient)
```

Create the optimizer

The optimizer is responsible for updating the weights, biases, beta, and gamma. In the code below, the sample code project creates an optimizer using the Adam algorithm:

```
static var adam = BNNS.AdamOptimizer(learningRate: 0.01,
                                      timeStep: 1,
                                      gradientScale: 1,
                                      regularizationScale: 0.01,
                                      gradientClipping: .byValue(bounds: -0.5 ... 0.5
                                      regularizationFunction: BNNSOptimizerRegulariza
```

Perform a backward pass and optimization step on the fully connected layer

The sample code project performs the backward pass in reverse order to the forward pass. Therefore, the sample's first step is to call `applyBackward` on the fully connected layer, and perform an optimization step on its weights.

The `applyBackward` call on the fully connected layer generates an input gradient that acts as the output gradient for the pooling layer's backward apply, and a weights gradient that passes to the fully connected optimizer step.

```
try fullyConnectedLayer.applyBackward(
    batchSize: batchSize,
    input: poolingOutput,
    output: fullyConnectedOutput,
    outputGradient: lossInputGradient,
    generatingInputGradient: fullyConnectedInputGradient,
    generatingWeightsGradient: fullyConnectedWeightGradient)
```

Perform a backward pass on the pooling layer

The backward pass on the pooling layer generates an input gradient that's the output gradient to the backward apply of the fused layer.

```
try poolingLayer.applyBackward(
    batchSize: batchSize,
    input: batchNormOutput,
    output: poolingOutput,
    outputGradient: fullyConnectedInputGradient,
    generatingInputGradient: poolingInputGradientArray)
```

Perform a backward pass and optimization step on the fused layer

The sample calls `applyBackward` on the fused layer. This performs an optimization step on the convolution layer's weights and bias, and the normalization layer's beta and gamma.

```
let gradientParameters = [convolutionWeightGradient,
                          convolutionBiasGradient,
                          batchNormBetaGradient,
                          batchNormGammaGradient]

try fusedConvBatchNormLayer.applyBackward(
    batchSize: batchSize,
    input: input,
    output: batchNormOutput,
    outputGradient: poolingInputGradientArray,
    generatingInputGradient: convolutionInputGradient,
    generatingParameterGradients: gradientParameters)
```

The code below performs the optimization step:

```
try adam.step(
    parameters: [fullyConnectedWeights,
                 convolutionWeights, convolutionBias,
                 batchNormBeta, batchNormGamma],
    gradients: [fullyConnectedWeightGradient,
                convolutionWeightGradient, convolutionBiasGradient,
                batchNormBetaGradient, batchNormGammaGradient],
    accumulators: [fullyConnectedWeightAccumulator1,
                   convolutionWeightAccumulator1, convolutionBiasAccumulator1,
                   batchNormBetaAccumulator1, batchNormGammaAccumulator1,
                   fullyConnectedWeightAccumulator2,
                   convolutionWeightAccumulator2, convolutionBiasAccumulator2,
                   batchNormBetaAccumulator2, batchNormGammaAccumulator2],
    filterParameters: filterParameters)
```

After the app completes all the optimization steps for this iteration, it increments the optimizer time step.

```
adam.timeStep += 1
```

Evaluate the neural network

The sample iterates over the forward, loss, backward, and optimization steps, and with each iteration, the trend of the loss is to reduce. The following graph shows the loss, as a solid stroke, decreasing during training:

The code in the sample defines a maximum number of iterations. Additionally, it calculates a moving average of recent loss values, which appear as a dashed stroke in the graph above. At each iteration, the sample checks whether the recent average loss is below that threshold, and, if it is, it breaks from the training phase early.

```
let maximumIterationCount = 1000

// The `recentLosses` array contains the last `recentLossesCount` losses.
let recentLossesCount = 20
var recentLosses = [Float]()

// The `averageRecentLossThreshold` constant defines the loss threshold
// at which to consider the training phase complete.
let averageRecentLossThreshold = Float(0.125)

for epoch in 0 ..< maximumIterationCount {
    if epoch == 500 {
        adam.learningRate /= 10
    }

    generateInputAndLabels()
    forwardPass()
    computeLoss()

    guard let loss = lossOutput.makeArray(of: Float.self,
                                          batchSize: 1)?.first else {
        print("Unable to calculate loss.")
        return
    }

    if recentLosses.isEmpty {
        recentLosses = [Float](repeating: loss,
                               count: recentLossesCount)
    }
```

```
    recentLosses[epoch % recentLossesCount] = loss
    let averageRecentLoss = vDSP.mean(recentLosses)

    if epoch % 10 == 0 {
        print("Epoch \(epoch): \(loss) : \(averageRecentLoss)")
    }

    if averageRecentLoss < averageRecentLossThreshold {
        print("Recent average loss: \(averageRecentLoss), breaking at epoch \(epoch)
        break
    }

    backwardPass()
}
```

After the training phase completes, the sample calculates the accuracy of the network over a new dataset. It then creates a new batch of random digits and runs a forward pass of the network.

```
try fusedConvBatchNormLayer.apply(batchSize: batchSize,
                                  input: input,
                                  output: batchNormOutput,
                                  for: .inference)

try poolingLayer.apply(batchSize: batchSize,
                       input: batchNormOutput,
                       output: poolingOutput)

try fullyConnectedLayer.apply(batchSize: batchSize,
                              input: poolingOutput,
                              output: fullyConnectedOutput)
```

Finally, the app evaluates the accuracy of the network by comparing the values in the fully connected layer's output to the one-hot labels. For example, when the recognized digit is 3, one-hot labels contain the values `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`, and values in the fully connected layer's output might be as follows:

```
[-2.51, -3.62, -0.10, 8.52, -0.42, 5.11, -1.65, 1.34,  0.82, -2.77]
[-3.94, -2.74, -0.30, 8.39, -1.45, 6.02, -0.66, 3.25,  0.49, -3.19]
[-2.51, -2.77, -0.77, 8.41, -0.82, 4.87, -0.37, 2.32, -0.49, -3.05]
[-3.01, -2.79,  0.48, 7.95, -2.57, 4.55, -1.05, 1.67,  1.38, -1.43]
[-2.48, -1.59, -0.97, 7.59, -2.52, 4.00,  0.95, 4.02, -2.10, -1.62]
```

Note that in each case, the highest value in the fully connected layer's output is at index 3.

The following code performs that evaluation for each digit in the batch:

```
guard
    let fullyConnected = fullyConnectedOutput.makeArray(
        of: Float.self,
        batchSize: batchSize),
    let labels = oneHotLabels.makeArray(
        of: Float.self,
        batchSize: batchSize) else {
    fatalError("Unable to create arrays for evaluation.")
}

var correctCount = 0

for sample in 0 ..< batchSize {
    let offset = fullyConnectedOutputWidth * sample

    let fullyConnectedBatch = fullyConnected[offset ..< offset + fullyConnectedOutpu
    let predictedDigit = vDSP.indexOfMaximum(fullyConnectedBatch).0

    let oneHotLabelsBatch = labels[offset ..< offset + fullyConnectedOutputWidth]
    let label = vDSP.indexOfMaximum(oneHotLabelsBatch).0

    if label == predictedDigit {
        correctCount += 1
    }

    print("Sample \(sample) — digit: \(label) | prediction: \(predictedDigit)")
}
```

The evaluation function prints out something like the following:

```
Sample 0 — digit: 7 | prediction: 7
Sample 1 — digit: 5 | prediction: 5
Sample 2 — digit: 7 | prediction: 7
Sample 3 — digit: 7 | prediction: 7
Sample 4 — digit: 0 | prediction: 0
Sample 5 — digit: 8 | prediction: 8
Sample 6 — digit: 3 | prediction: 3
Sample 7 — digit: 6 | prediction: 6
```

```
Sample 8 — digit: 2 | prediction: 2
Sample 9 — digit: 7 | prediction: 7
[ ... ]
```

In this case, the neural network accurately predicts each ground truth digit.

See Also

Neural Networks

BNNS

Implement and run neural networks for training and inference.