Sample Code

Compressing an image using linear algebra

Reduce the storage size of an image using singular value decomposition (SVD).

Download

macOS 13.3+ | Xcode 14.0+

Overview

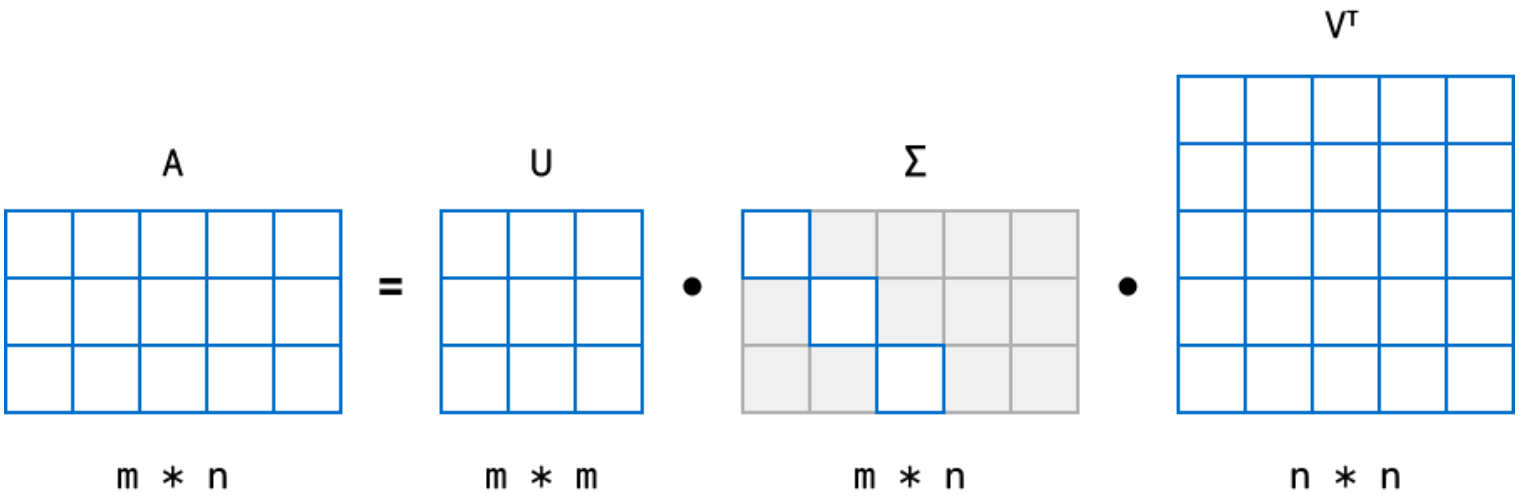
This sample code project decomposes an image into three factors using singular value decomposition (SVD). The sample compresses an image by computing the products of the factors submatrices. The image below shows two photographs. The first is the original image, and the second is the same image after the sample applies 10:1 compression:

Any *m* x *n* matrix, *A*, has an SVD factorization that decomposes it into three factors:

- The *m* x *m* matrix *U* that contains the left singular vectors of matrix *A*
- The *m* x *n* diagonal matrix *Σ* that contains the singular values of matrix *A*, arranged in descending order
- The *n* x *n* transposed matrix *V* that contains the right singular vectors of matrix *A*

The sample uses the Linear Algebra Package (LAPACK) function `sgesvdx_` to compute the SVD.

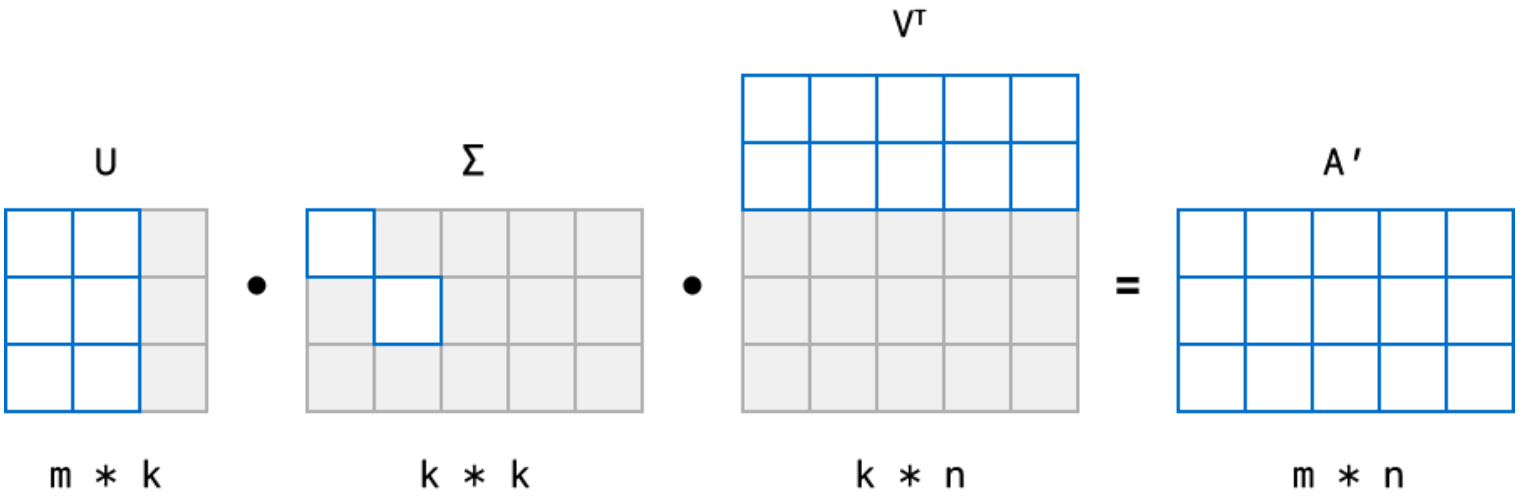
The figure below shows the SVD of a 5 x 3 matrix:



When matrix *A* contains image information, the magnitude of the singular values correlate to the visual significance of features in the image.

The sample reduces the storage size of the original image by returning the product of submatrices of *U*, *Σ*, and V^T. The sizes of the submatrices derive from the index of the first low singular value. The code in this sample defines that value as `k`.

For example, if the diagonal elements of *Σ* are `[2000, 1000, 24]` and you define `k` as 2, the app keeps `[2000, 1000]` and discards `24`. The following figure shows the matrix multiply function for this example, where the first two singular values contain significant values:



Create a matrix from the source image

The sample defines a `Matrix` structure that simplifies passing image data to Accelerate's linear algebra libraries.

```
public struct Matrix {
    /// The number of rows in the matrix.
    public let rowCount: Int

    /// The number of columns in the matrix.
    public let columnCount: Int

    /// The total number of elements in the matrix.
    public var count: Int {
        return rowCount * columnCount
    }

    /// A pointer to the matrix's underlying data.
    public var data: UnsafeMutableBufferPointer<Float> {
        get {
            return dataReference.data
        }
        set {
            dataReference.data = newValue
        }
    }

    /// A pointer to the matrix's underlying data reference.
    private var dataReference: MatrixDataReference

    /// An object that wraps the structure's data and provides deallocation when the
    private class MatrixDataReference {
        var data: UnsafeMutableBufferPointer<Float>

        init(data: UnsafeMutableBufferPointer<Float>) {
            self.data = data
        }

        deinit {
            self.data.deallocate()
        }
    }
}
```

The `Matrix` type provides an initializer that returns a new matrix that contains a 32-bit grayscale copy of the pixel values of a vImage buffer. The `init(cgImage:)` function passes `kvImageNoAllocate` to `vImageBuffer_InitWithCGImage(_:_:_:_:)` and uses the matrix's

memory allocation. This ensures that there are no extra bytes at the end of each row. See “Create floating point pixels to use with vDSP” in Finding the sharpest image in a sequence of captured images for more information on row byte padding.

```
/// The 32-bit planar image format that the `Matrix` type uses to
/// consume and produce `CGImage` instances.
private static var imageFormat = vImage_CGImageFormat(
    bitsPerComponent: 32,
    bitsPerPixel: 32,
    colorSpace: CGColorSpaceCreateDeviceGray(),
    bitmapInfo: CGBitmapInfo(rawValue:
                             kCGBitmapByteOrder32Host.rawValue |
                             CGBitmapInfo.floatComponents.rawValue |
                             CGImageAlphaInfo.none.rawValue))!

/// Converts the specified image to 32-bit planar and returns a new matrix
/// that contains that image data.
public init?(cgImage: CGImage) {

    self.init(rowCount: cgImage.height,
              columnCount: cgImage.width)

    // Create a `vImage_Buffer` that shares data with `self`.
    var tmpBuffer = vImage_Buffer(
        data: self.data.baseAddress,
        height: vImagePixelCount(self.rowCount),
        width: vImagePixelCount(self.columnCount),
        rowBytes: self.columnCount * MemoryLayout<Float>.stride)

    let error = vImageBuffer_InitWithCGImage(
        &tmpBuffer,
        &Matrix.imageFormat,
        [0, 0, 0, 0],
        cgImage,
        vImage_Flags(kvImageNoAllocate))

    if error != kvImageNoError {
        return nil
    }
}
```

The following code creates a matrix instance from a Core Graphics image:

```
guard
    let sourceCGImage = image.cgImage(forProposedRect: nil,
                                      context: nil,
                                      hints: nil),
    let sourceImageMatrix = Matrix(cgImage: sourceCGImage)  else {
    fatalError("Error initializing `SVDImageCompressor` instance.")
}
```

Create the factor matrices

The `Matrix` type provides an initializer that returns a new zero-filled matrix.

```
/// Returns a zero-filled matrix.
public init(rowCount: Int,
            columnCount: Int) {

    let count = rowCount * columnCount

    let start = UnsafeMutablePointer<Float>.allocate(capacity: count)

    let buffer = UnsafeMutableBufferPointer(start: start,
                                            count: count)
    buffer.initialize(repeating: 0)

    self.rowCount = rowCount
    self.columnCount = columnCount
    self.dataReference = MatrixDataReference(data: buffer)
}
```

The following code creates the three matrices that represent the factors:

```
/// The _U_ in _A = U * Σ * Vᵀ_.
let u = Matrix(rowCount: a.rowCount,
               columnCount: k)

/// The diagonal values of _Σ_ in _A = U * Σ * Vᵀ_.
let sigma = Matrix(rowCount: min(a.rowCount, a.columnCount),
                   columnCount: 1)

/// The _Vᵀ_ in _A = U * Σ * Vᵀ_.
```

```
let vt = Matrix(rowCount: k,
                columnCount: a.columnCount)
```

Define the SVD options

The sample requires fully populated *U* and V^T matrices, and defines the `JOBU` and `JOBVT` parameters that it passes to `sgesvdx_` as `V`. In order to specify that `sgesvdx_` returns a specified number of singular values, the sample defines the `RANGE` parameter as `I`.

```
var JOBU = Int8("V".utf8.first!)
var JOBVT = Int8("V".utf8.first!)
var RANGE = Int8("I".utf8.first!)
```

Create the workspaces

Before computing the SVD, the sample performs a workspace query to calculate the optimal size of the workspace that `sgesvdx_` requires. The following code specifies `LWORK` as `–1` and passes a pointer to a single `Float` to the `WORK` parameter.

```
var minusOne = __LAPACK_int(-1)
var workspaceDimension = Float()
sgesvdx_(&JOBU,
         &JOBVT,
         &RANGE,
         &m,
         &n,
         aCopy.baseAddress,
         &lda,
         &vl,
         &vu,
         &il,
         &iu,
         &ns,
         sigma.data.baseAddress,
         u.data.baseAddress,
         &ldu,
         vt.data.baseAddress,
         &ldvt,
         &workspaceDimension,
         &minusOne,
         iwork,
```

```
        &info)
```

On return, `workspaceDimension` contains the optimal size for the workspace.

The following code allocates the memory that the SVD routine uses as the workspace:

```
var lwork = __LAPACK_int(workspaceDimension)

let workspace = UnsafeMutablePointer<Float>.allocate(capacity: Int(lwork))
defer {
    workspace.deallocate()
}
```

Use LAPACK to compute the SVD

To avoid multiple nested calls to `withUnsafePointer(to:_:)`, the sample declares variables that shadow some of the matrix properties to pass to the SVD routine as `UnsafePointer` structures.

```
var m = __LAPACK_int(a.m)
var n = __LAPACK_int(a.n)
var lda = __LAPACK_int(a.m)

var ldu = __LAPACK_int(u.m)
var ldvt = __LAPACK_int(vt.m)
```

The samples creates the `iwork` integer array with a count of 12 times the minimum dimension of matrix *A*.

```
let iwork = UnsafeMutablePointer<__LAPACK_int>.allocate(capacity: 12 * Int(min(m, n)
defer {
    iwork.deallocate()
}
```

The sample calls the LAPACK function for a second time to compute the SVD.

```
// Compute `iu - il + 1` singular values.
sgesvdx_(&JOBU,
         &JOBVT,
         &RANGE,
```

```
        &m,
        &n,
        aCopy.baseAddress,
        &lda,
        &vl,
        &vu,
        &il,
        &iu,
        &ns,
        sigma.data.baseAddress,
        u.data.baseAddress,
        &ldu,
        vt.data.baseAddress,
        &ldvt,
        workspace,
        &lwork,
        iwork,
        &info)
```

On return, the matrices `sigma`, u, and `vt` contain the SVD result.

Convert the singular values vector to a matrix

The `sgesvdx_` function computes the singular values as a vector. To convert the vector of k diagonal values to a k `*` k diagonal matrix, the `Matrix` type provides an initializer that returns a new matrix from diagonal values.

```
/// Returns a column-major matrix with the specified diagonal elements.
public init<C>(diagonal: C,
               rowCount: Int,
               columnCount: Int)
where
C: Collection,
C.Index == Int,
C.Element == Float {

    self.init(rowCount: rowCount,
              columnCount: columnCount)

    for i in 0 ..< min(rowCount, columnCount, diagonal.count) {
        self[i * rowCount + i] = diagonal[i]
    }
```

```
}
```

The following code creates a k * k diagonal matrix from the SVD sigma values:

```
let sigma = Matrix(diagonal: svdResult.sigma.data,
                   rowCount: Int(k),
                   columnCount: Int(k))
```

Multiply the factors

The `Matrix` type provides a static function that wraps `cblas_sgemm(_:_:_:_:_:_:_:_:_:_:_:_:_:)` to multiply two matrices.

```
public static func multiply(a: Matrix,
                            b: Matrix,
                            c: Matrix,
                            k: Int32? = nil) {

    cblas_sgemm(CblasColMajor,
                CblasNoTrans, CblasNoTrans,
                a.m,
                b.n,
                k ?? b.m,
                1,
                a.data.baseAddress, a.m,
                b.data.baseAddress, b.m,
                0,
                c.data.baseAddress, c.m)
}
```

The sample uses the matrix multiply function to recreate matrix *A* from the SVD factors.

```
/// The matrix that receives `u * sigma`.
let u_sigma = Matrix(rowCount: svdResult.u.rowCount,
                     columnCount: sigma.columnCount)

Matrix.multiply(a: svdResult.u,
                b: sigma,
                c: u_sigma)
```

```
/// The matrix that receives `u * sigma * vᵀ`.
let u_sigma_vt = Matrix(rowCount: u_sigma.rowCount,
                        columnCount: svdResult.vt.columnCount)

Matrix.multiply(a: u_sigma,
                b: svdResult.vt,
```

Create a Core Graphics image from the product of the factors

The `cgImage` computed property returns a `CGImage` instance from a matrix's data. The computed property creates a temporary vImage buffer that shares its data with the matrix and uses the same planar 32-bit `vImage_CGImageFormat` that the `init(cgImage:)` initializer uses to convert a Core Graphics image to a matrix.

```
/// Returns a 32-bit per pixel, grayscale `CGImage`instance of the matrix's data.
public var cgImage: CGImage? {

    let tmpBuffer = vImage_Buffer(
        data: self.data.baseAddress!,
        height: vImagePixelCount(self.rowCount),
        width: vImagePixelCount(self.columnCount),
        rowBytes: self.columnCount * MemoryLayout<Float>.stride)

    return try? tmpBuffer.createCGImage(format: Matrix.imageFormat)
}
```

See Also

Linear Algebra

{} Solving systems of linear equations with LAPACK

Select the optimal LAPACK routine to solve a system of linear equations.

Finding an interpolating polynomial using the Vandermonde method

Use LAPACK to solve a linear system and find an interpolating polynomial to construct new points between a series of known data points.

BLAS

Perform common linear algebra operations with Apple's implementation of the Basic Linear Algebra Subprograms (BLAS).