Neural Networks

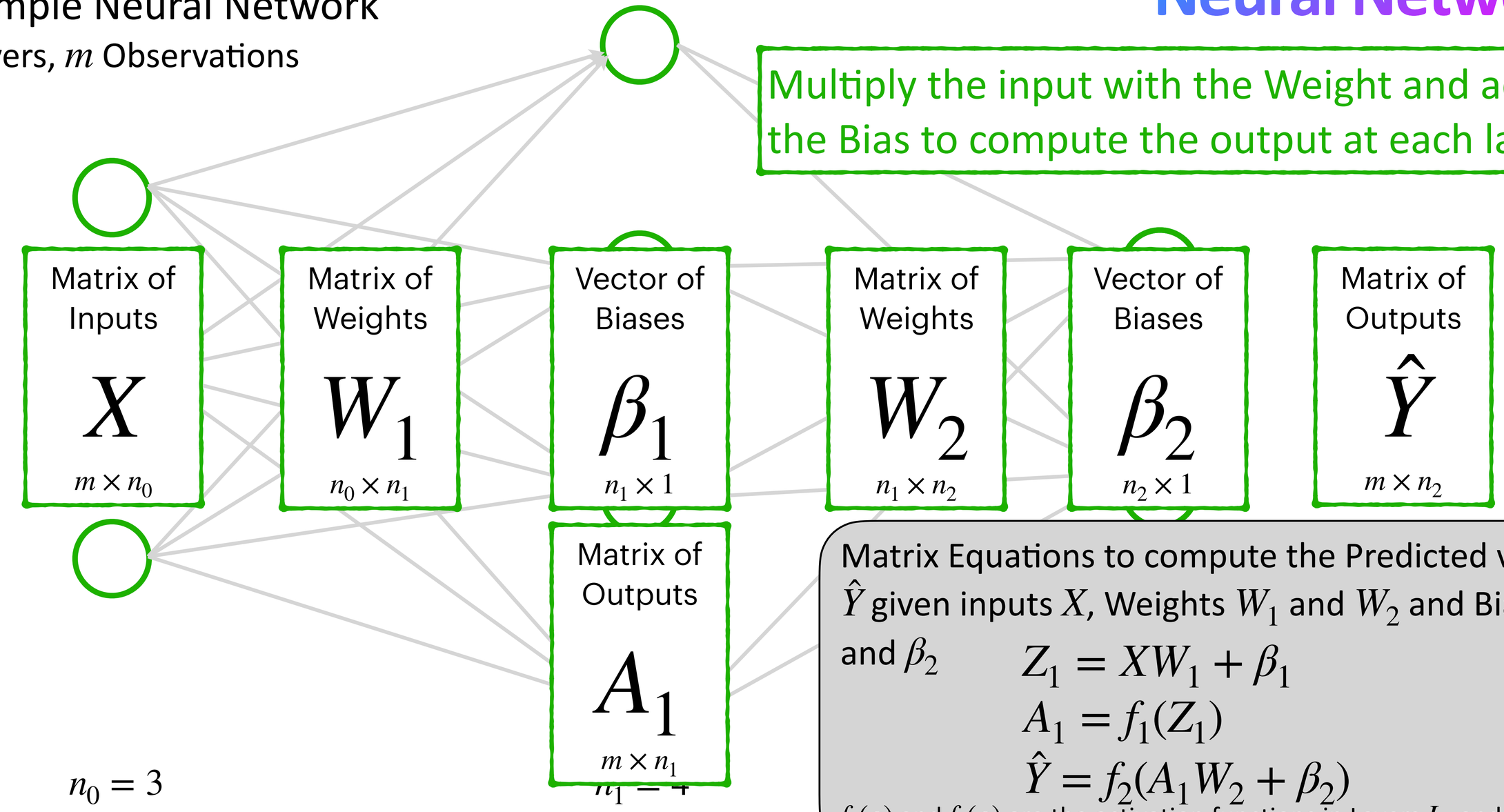
Implementing a Neural Network in Go

Rahul Singh
rsingh@arrsingh.com

Neural Networks

A Simple Neural Network

3 Layers, m Observations



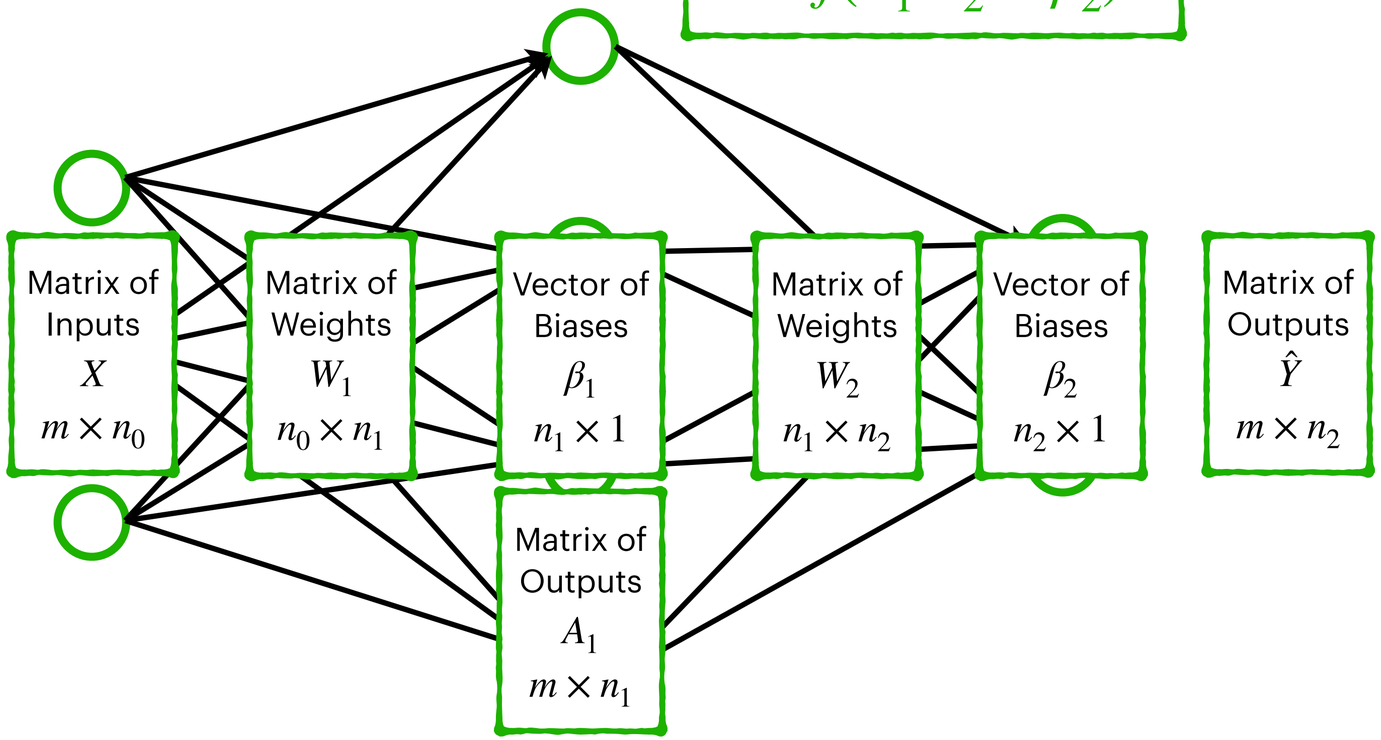
Matrix Equations to compute the Predicted values $\hat{Y}$ given inputs X, Weights W_1 and W_2 and Biases β_1 and β_2

$$Z_1 = XW_1 + \beta_1$$
$$A_1 = f_1(Z_1)$$
$$\hat{Y} = f_2(A_1W_2 + \beta_2)$$

$f_1(g)$ and $f_2(g)$ are the activation functions in Layers L_1 and L_2

Neural Networks

$$Z_1 = XW_1 + \beta_1$$
$$A_1 = f(Z_1)$$
$$\hat{Y} = f(A_1 W_2 + \beta_2)$$



$n_0 = 3$
Input Layer (L_0)

$n_1 = 4$
Hidden Layer (L_1)

$n_2 = 2$
Output Layer (L_2)

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute…

$$Z_1 = XW_1 + \beta_1$$
$$A_1 = f(Z_1)$$
$$\hat{Y} = f(A_1 W_2 + \beta_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$
$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Neural Networks

Let’s go through all the pieces we need to build

Parameter Initialization

```
// Parameters in a neural network
// Thre layers - L0 (Input), L1 (Hidden) L2 (Output)
// n0 = neurons in the input layer (L0)
// n1 = neurons in the hidden layer (L1)
// n2 = neurons in the output layer (L2)
type NNParams struct {
    // Matrix of Weights n0 x n1
    W1 *mat.Dense
    // Matrix of Weights n1 x n2
    W2 *mat.Dense
    // Vector of biases n1 x 1
    B1 *mat.Dense
    // Vector of biases n2 x 1
    B2 *mat.Dense
}
```

```
func (p *NNParams) Initialize(n0, n1, n2 int) *NNParams {
    p.W1 = mat.NewDense(n0, n1, nil)
    p.W1.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W1)

    p.B1 = mat.NewDense(n1, 1, nil)

    p.W2 = mat.NewDense(n1, n2, nil)
    p.W2.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W2)

    p.B2 = mat.NewDense(n2, 1, nil)

    return p
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Neural Networks

Parameter Initialization

```go
// Parameters in a neural network
// Thre layers - L0 (Input), L1 (Hidden) L2 (Output)
// n0 = neurons in the input layer (L0)
// n1 = neurons in the hidden layer (L1)
// n2 = neurons in the output layer (L2)
type NNParams struct {
    // Matrix of Weights n0 x n1
    W1 *mat.Dense
    // Matrix of Weights n1 x n2
    W2 *mat.Dense
    // Vector of biases n1 x 1
    B1 *mat.Dense
    // Vector of biases n2 x 1
    B2 *mat.Dense
}
```

```go
func (p *NNParams) Initialize(n0, n1, n2 int) *NNParams {
    p.W1 = mat.NewDense(n0, n1, nil)
    p.W1.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W1)

    p.B1 = mat.NewDense(n1, 1, nil)

    p.W2 = mat.NewDense(n1, n2, nil)
    p.W2.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W2)

    p.B2 = mat.NewDense(n2, 1, nil)

    return p
}
```

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Neural Networks

Parameter Initialization

```go
// Parameters in a neural network
// Thre layers - L0 (Input), L1 (Hidden) L2 (Output)
// n0 = neurons in the input layer (L0)
// n1 = neurons in the hidden layer (L1)
// n2 = neurons in the output layer (L2)
type NNParams struct {
    // Matrix of Weights n0 x n1
    W1 *mat.Dense
    // Matrix of Weights n1 x n2
    W2 *mat.Dense
    // Vector of biases n1 x 1
    B1 *mat.Dense
    // Vector of biases n2 x 1
    B2 *mat.Dense
}
```

```go
func (p *NNParams) Initialize(n0, n1, n2 int) *NNParams {
    p.W1 = mat.NewDense(n0, n1, nil)
    p.W1.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W1)

    p.B1 = mat.NewDense(n1, 1, nil)

    p.W2 = mat.NewDense(n1, n2, nil)
    p.W2.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W2)

    p.B2 = mat.NewDense(n2, 1, nil)

    return p
}
```

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Weight Matrices are initialized with a random sample from a normal distribution. The weights are multiplied by a small value (0.001) to avoid gradient explosion.

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Neural Networks

Parameter Initialization

```go
// Parameters in a neural network
// Thre layers - L0 (Input), L1 (Hidden) L2 (Output)
// n0 = neurons in the input layer (L0)
// n1 = neurons in the hidden layer (L1)
// n2 = neurons in the output layer (L2)
type NNParams struct {
    // Matrix of Weights n0 x n1
    W1 *mat.Dense
    // Matrix of Weights n1 x n2
    W2 *mat.Dense
    // Vector of biases n1 x 1
    B1 *mat.Dense
    // Vector of biases n2 x 1
    B2 *mat.Dense
}
```

```go
func (p *NNParams) Initialize(n0, n1, n2 int) *NNParams {
    p.W1 = mat.NewDense(n0, n1, nil)
    p.W1.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W1)

    p.B1 = mat.NewDense(n1, 1, nil)

    p.W2 = mat.NewDense(n1, n2, nil)
    p.W2.Apply(func(i, j int, v float64) float64 {
        return rand.NormFloat64() * 0.001
    }, p.W2)

    p.B2 = mat.NewDense(n2, 1, nil)

    return p
}
```

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Weight Matrices are initialized with a random sample from a normal distribution. The weights are multiplied by a small value (0.001) to avoid gradient explosion.

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Bias Matrices are initialized with zeros

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)

```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the same dimensions")
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$Z_2 = A_1W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial\beta_2}cost \quad \frac{\partial}{\partial W_2}cost \quad \frac{\partial}{\partial\beta_1}cost \quad \frac{\partial}{\partial W_1}cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial\beta_2}cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2}cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial\beta_1}cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1}cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the same dimensions")
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

$, W_2, \beta_1, \beta_2$

$= tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1...

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the sam
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Compute the Softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$, W_2, \beta_1, \beta_2$

$= tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

ivative of the cost

$, W_2, \beta_1, W_1 \ldots$

$\frac{\partial}{\beta_1} cost \qquad \frac{\partial}{\partial W_1} cost$

$\partial\beta_2$ ate

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the sam
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Compute the Softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

1. Compute e^{z_i} for each class

$$\ldots, W_2, \beta_1, \beta_2$$

$$= tanh(Z_1)$$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

ivative of the cost

$$\ldots, W_2, \beta_1, W_1 \ldots$$

$$\frac{\partial}{\beta_1} cost \qquad \frac{\partial}{\partial W_1} cost$$

ate

$$\times learning\_rate$$

$$\frac{\partial}{\partial W_2}$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the sam
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Compute the Softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

1. Compute e^{z_i} for each class

2. Compute the sum of the exponents

$, W_2, \beta_1, \beta_2$

$= tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

ivative of the cost

$, W_2, \beta_1, W_1 \ldots$

$\frac{\partial}{\beta_1} cost \qquad \frac{\partial}{\partial W_1} cost$

ate

$\times learning\_rate$

$\frac{\partial}{\partial \beta_1}$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the sam
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Compute the Softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

1. Compute e^{z_i} for each class
2. Compute the sum of the exponents
3. Set the result as $\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

For more details see the tutorial on Activation Functions

$, W_2, \beta_1, \beta_2$

$= tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

ivative of the cost

$, W_2, \beta_1, W_1 \ldots$

$\frac{\partial}{\beta_1} cost \qquad \frac{\partial}{\partial W_1} cost$

ate

$\times learning\_rate$

$\partial\beta_1$

rning_rate

ues for $\beta_1, W_1, \beta_2, W_2$

$W_2 - step\_size_{W_2}$

$W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Activation Functions (tanh & softmax)


```go
func Tanh(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Tanh: dimension mismatch - x and result must have the same dimensions")
    }

    result.Apply(func(i, j int, v float64) float64 {
        return math.Tanh(v)
    }, x)
}
```

Apply `math.Tanh` to each element of the matrix X and set it in the result matrix

$$, W_2, \beta_1, \beta_2$$
$$= tanh(Z_1)$$
$$Z_2 = A_1 W_2 + \beta_2$$
$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1 \ldots$

$$\frac{\partial}{\partial \beta_1} cost \qquad \frac{\partial}{\partial W_1} cost$$

```go
func Softmax(x *mat.Dense, result *mat.Dense) {
    xRows, xCols := x.Dims()
    rRows, rCols := result.Dims()

    if xRows != rRows || xCols != rCols {
        panic("Softmax: dimension mismatch - x and result must have the same")
    }

    expVals := make([]float64, xCols)

    for i := range xRows {
        maxVal := x.At(i, 0)
        for j := 1; j < xCols; j++ {
            if val := x.At(i, j); val > maxVal {
                maxVal = val
            }
        }

        var sum float64
        for j := range xCols {
            expVals[j] = math.Exp(x.At(i, j) - maxVal)
            sum += expVals[j]
        }

        for j := range xCols {
            result.Set(i, j, expVals[j]/sum)
        }
    }
}
```

Compute the Softmax function:

Numerical Stability: We calculate the max value and subtract it from each exponent to avoid overflow due to exponents of large values.

If z_1 is large then e^{z_i} will overflow. However $e^{z_i - max(z_i)}$ will be not overflow because the largest value will be $e^0 = 1$ and the others will lie between 0 and 1 (for $e^{negative}$)

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$
$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (derivative of tanh)

```go
func DerivativeTanh(activationOutput *mat.Dense, result *mat.Dense) {
    aRows, aCols := activationOutput.Dims()
    rRows, rCols := result.Dims()

    if aRows != rRows || aCols != rCols {
        panic(`DerivativeTanh: dimension mismatch -
              activationOutput and result must have the same dimensions`)
    }

    result.Apply(func(i, j int, v float64) float64 {
        return 1.0 - v*v
    }, activationOutput)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Activation Functions (derivative of tanh)

```go
func DerivativeTanh(activationOutput *mat.Dense, result *mat.Dense) {
    aRows, aCols := activationOutput.Dims()
    rRows, rCols := result.Dims()

    if aRows != rRows || aCols != rCols {
        panic(`DerivativeTanh: dimension mismatch -
              activationOutput and result must have the same dimensions`)
    }

    result.Apply(func(i, j int, v float64) float64 {
        return 1.0 - v*v
    }, activationOutput)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost

$W_2, \beta_1, W_1 \cdots$

$ost \quad \frac{\partial}{\partial W_1} cost$

Compute the Partial Derivative of the Tanh activation function:

$$\frac{\partial}{\partial x} tanh(x) = 1 - tanh^2(x)$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Activation Functions (derivative of tanh)

```go
func DerivativeTanh(activationOutput *mat.Dense, result *mat.Dense) {
    aRows, aCols := activationOutput.Dims()
    rRows, rCols := result.Dims()

    if aRows != rRows || aCols != rCols {
        panic(`DerivativeTanh: dimension mismatch -
              activationOutput and result must have the same dimensions`)
    }

    result.Apply(func(i, j int, v float64) float64 {
        return 1.0 - v*v
    }, activationOutput)
}
```

Compute the Partial Derivative of the Tanh activation function:

$$\frac{\partial}{\partial x} tanh(x) = 1 - tanh^2(x)$$

Each element of activationOutput, v (at i, j) is $tanh$, so the partial derivative is simply $1 - v^2$

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost

$W_2, \beta_1, W_1 \cdots$

ost $\frac{\partial}{\partial W_1} cost$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)

```go
// CostFunction computes the categorical cross-entropy cost for multi-class classification.
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples, n2 = output classes
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number of output classes)
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij}\, log_e \hat{y}_{ij}$$

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij}\, log_e \hat{y}_{ij}$$

Iterate over all the observations ($i = 1..m$)

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost

$W_1 \cdots$

$\partial\beta_2 \quad \partial W_2 \quad \partial\beta_1 \quad \partial W_1 \quad -cost$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial\beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial\beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij} log_e \hat{y}_{ij}$$

Iterate over all the observations ($i = 1..m$)

Iterate over all the output classes ($j = 1..K$)

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1 W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost

$W_1 \cdots$

$-cost$

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij} log_e \hat{y}_{ij}$$

Iterate over all the observations ($i = 1..m$)

Iterate over all the output classes ($j = 1..K$)

Sum all values of $y_j log(\hat{y}_j)$

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1 W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost

$W_1 \cdots$

$-cost$

$t \times learning\_rate$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij} log_e \hat{y}_{ij}$$

Iterate over all the observations ($i = 1..m$)

Iterate over all the output classes ($j = 1..K$)

Sum all values of $y_j log(\hat{y}_j)$

Inner loop accumulates the sum of classes and outer loop accumulates the sum of samples

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1 W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost

$W_1 \cdots$

$-cost$

$t \times learning\_rate$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Cost Function (Categorical Cross Entropy)


```go
// CostFunction computes the categorical cross-entropy cost for multi-
// Formula: Cost = -1/m × Σ(i=1..m) Σ(j=1..K) [yi × log(y_hat_j)]
// Parameters:
//   - yhat: predictions from the network (m × n2), where m = samples,
//   - y: true labels (m × n2), typically one-hot encoded
//
// Returns: scalar cost value
func CostFunction(yhat *mat.Dense, y *mat.Dense) float64 {
    // Get dimensions: rows = m (number of samples), cols = n2 (number
    rows, cols := y.Dims()

    // Convert m to float64 for division
    m := float64(rows)

    // Accumulator for the sum of y × log(ŷ) across all elements
    var sum float64

    // Iterate over all training examples
    for i := range rows {
        // Iterate over all output classes
        for j := range cols {
            // Get true label value for sample i, class j
            yVal := y.At(i, j)

            // Get predicted probability for sample i, class j
            aVal := yhat.At(i, j)

            // Add y × log(ŷ) to the sum (element-wise multiplication)
            sum += yVal * math.Log(aVal)
        }
    }

    // Apply the cost formula: -1/m × sum
    cost := -(1.0 / m) * sum
    return cost
}
```

Cost Function:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{ij} log_e \hat{y}_{ij}$$

Iterate over all the observations ($i = 1..m$)

Iterate over all the output classes ($j = 1..K$)

Sum all values of $y_j log(\hat{y}_j)$

Inner loop accumulates the sum of classes and outer loop accumulates the sum of samples

Negate and divide by m to compute average cost

For more details see the tutorial on Activation Functions

l values for $W_1, W_2, \beta_1, \beta_2$

$= XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$= A_1 W_2 + \beta_2$

$= softmax(Z_2)$

artial derivative of the cost

$W_1 \cdots$

$-cost$

$t \times learning\_rate$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

$Z_1 = XW_1$

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

$Z_1 = XW_1$

$Z_1 = XW_1 + \beta_1$

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

$Z_1 = XW_1$

$Z_1 = XW_1 + \beta_1$

$A_1 = tanh(Z_1)$

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost

on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

ute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$= \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

$Z_1 = XW_1$

$Z_1 = XW_1 + \beta_1$

$A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2$

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost

on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$

ute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$= \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$ize_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//  Z₁ = XW₁ + β₁
//  A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//  Z₂ = A₁W₂ + β₂
//  Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//   - x: input data (m × n0), where m = samples, n0 = input features
//   - params: network parameters (W1, B1, W2, B2)
//   - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$ $Z_2 = A_1W_2 + \beta_2$ $\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$= \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$ize_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$\frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Forward Propagation

```go
// ForwardPropagation computes the forward pass through the neural network.
// Equations:
//
//   Z₁ = XW₁ + β₁
//   A₁ = f₁(Z₁)  [where f₁ is the activation function for layer 1, using Tanh]
//   Z₂ = A₁W₂ + β₂
//   Ŷ = f₂(Z₂)   [where f₂ is the activation function for layer 2, using Softmax]
//
// Parameters:
//    - x: input data (m × n0), where m = samples, n0 = input features
//    - params: network parameters (W1, B1, W2, B2)
//    - cache: pre-allocated cache to store intermediate values
//
// The function populates the cache with all intermediate computations
func ForwardPropagation(x *mat.Dense, params *NNParams, cache *Cache) {
    m, _ := x.Dims()
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Layer 1: Z₁ = XW₁ + β₁
    // Compute XW₁
    cache.Z1.Mul(x, params.W1)

    // Add bias β₁ to each row (broadcasting B1 across all samples)
    for i := range m {
        for j := range n1 {
            cache.Z1.Set(i, j, cache.Z1.At(i, j)+params.B1.At(j, 0))
        }
    }

    // Layer 1 Activation: A₁ = f₁(Z₁) = Tanh(Z₁)
    Tanh(cache.Z1, cache.A1)

    // Layer 2: Z₂ = A₁W₂ + β₂
    // Compute A₁W₂
    cache.Z2.Mul(cache.A1, params.W2)

    // Add bias β₂ to each row (broadcasting B2 across all samples)
    for i := range m {
        for j := range n2 {
            cache.Z2.Set(i, j, cache.Z2.At(i, j)+params.B2.At(j, 0))
        }
    }

    // Layer 2 Activation: Ŷ = f₂(Z₂) = Softmax(Z₂)
    Softmax(cache.Z2, cache.Yhat)
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$ $Z_2 = A_1W_2 + \beta_2$ $\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost on w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$= \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$ize_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$\frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

ize_{β_2} $W_2 = W_2 - step\_size_{W_2}$

ize_{β_1} $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Gradients) {
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Gradients) {
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

Layer 2 Gradients…

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Gradients) {
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

Layer 2 Gradients…

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Neural Networks

$\beta_2, W_2, \beta_1, W_1$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$Z_2 = A_1 W_2 + \beta_2$

$\hat{Y} = softmax(Z_2)$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients…

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Neural Networks

$\beta_2, \beta_1, \beta_2$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients…

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

Neural Networks

$\beta_2, W_2, \beta_1, W_1$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1…

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients...

Layer 1 Gradients...

Neural Networks

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

$\beta_2, W_2, \beta_1, W_2$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1...

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

p 4: Compute step sizes...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients...

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

Layer 1 Gradients...

Compute $\frac{\partial}{\partial A_1} cost = (\frac{\partial}{\partial Z_2} cost) W_2^T$

Neural Networks

$\beta_2, W_2, \beta_1, W_1$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1...

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

p 4: Compute step sizes...

$\times\ learning\_rate$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients…

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

Layer 1 Gradients…

Compute $\frac{\partial}{\partial A_1} cost = (\frac{\partial}{\partial Z_2} cost) W_2^T \times learning\_rate$

Compute $\frac{\partial}{\partial Z_1} cost = \frac{\partial}{\partial A_1} cost \odot \left[\frac{\partial}{\partial Z_1} f_1(Z_1)\right] = (\frac{\partial}{\partial Z_2} cost) W_2^T \odot (1 - A_1^2)$

This is the Hadamard product (element wise product) not a matrix multiplication

Neural Networks

$\beta_2, \beta_1, \beta_2$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1…

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

p 4: Compute step sizes…

$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients...

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

Layer 1 Gradients...

Compute $\frac{\partial}{\partial A_1} cost = (\frac{\partial}{\partial Z_2} cost) W_2^T$

Compute $\frac{\partial}{\partial Z_1} cost = \frac{\partial}{\partial A_1} cost \odot \left[\frac{\partial}{\partial Z_1} f_1(Z_1)\right] = (\frac{\partial}{\partial Z_2} cost) W_2^T \odot (1 - A_1^2)$

This is the Hadamard product (element wise product) not a matrix multiplication

Compute $\frac{\partial}{\partial W_1} cost = \frac{1}{m} X^T \frac{\partial}{\partial Z_1} cost$

Neural Networks

$\beta_2, \beta_1, \beta_2$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1...

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

p 4: Compute step sizes...

$\times learning\_rate$

$\times learning\_rate$

values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Back Propagation

```go
// Parameters:
//   - x: input data (m × n0)
//   - y: true labels (m × n2)
//   - params: network parameters
//   - forwardCache: cached values from forward propagation
//   - grads: pre-allocated gradients struct to store computed gradients
func BackPropagation(x *mat.Dense, y *mat.Dense, params *NNParams, forwardCache *Cache, grads *Grad
    m, _ := x.Dims()
    mFloat := float64(m)
    _, n1 := params.W1.Dims()
    _, n2 := params.W2.Dims()

    // Temporary matrices for intermediate computations
    dZ2 := mat.NewDense(m, n2, nil)
    dZ1 := mat.NewDense(m, n1, nil)
    dA1 := mat.NewDense(m, n1, nil)

    // Layer 2 gradients
    // dZ2 = A2 - Y (where A2 is Yhat from forward cache)
    dZ2.Sub(forwardCache.Yhat, y)

    // dW2 = (1/m) * A1ᵀ * dZ2
    grads.DW2.Mul(forwardCache.A1.T(), dZ2)
    grads.DW2.Scale(1.0/mFloat, grads.DW2)

    // dB2 = (1/m) * sum(dZ2, axis=1, keepdims=True)
    // Sum across rows (axis=1 means sum each row to get column vector)
    for j := range n2 {
        var sum float64
        for i := range m {
            sum += dZ2.At(i, j)
        }
        grads.DB2.Set(j, 0, sum/mFloat)
    }

    // Layer 1 gradients
    // dZ1 = dZ2 * W2ᵀ * derivative_tanh(A1)
    // First: dZ2 * W2ᵀ
    dA1.Mul(dZ2, params.W2.T())

    // Then: multiply element-wise by derivative_tanh(A1)
    // derivative_tanh(A1) = 1 - A1²
    for i := range m {
        for j := range n1 {
            a1Val := forwardCache.A1.At(i, j)
            dTanh := 1.0 - a1Val*a1Val
            dZ1.Set(i, j, dA1.At(i, j)*dTanh)
        }
    }

    // dW1 = (1/m) * Xᵀ * dZ1
    grads.DW1.Mul(x.T(), dZ1)
    grads.DW1.Scale(1.0/mFloat, grads.DW1)

    // dB1 = (1/m) * sum(dZ1, axis=1, keepdims=True)
    for j := range n1 {
        var sum float64
        for i := range m {
            sum += dZ1.At(i, j)
        }
        grads.DB1.Set(j, 0, sum/mFloat)
    }
}
```

Layer 2 Gradients...

Compute error from the output layer: $\frac{\partial}{\partial Z_2} cost = \hat{Y} - Y$

Compute $\frac{\partial}{\partial W_2} cost = \frac{\partial}{\partial W_2} Z_2 \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T \frac{\partial}{\partial Z_2} cost = \frac{1}{m} A_1^T (\hat{Y} - Y)$

Compute $\frac{\partial}{\partial \beta_2} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_2} cost$

Layer 1 Gradients...

Compute $\frac{\partial}{\partial A_1} cost = (\frac{\partial}{\partial Z_2} cost) W_2^T$

Compute $\frac{\partial}{\partial Z_1} cost = \frac{\partial}{\partial A_1} cost \odot \left[\frac{\partial}{\partial Z_1} f_1(Z_1)\right] = (\frac{\partial}{\partial Z_2} cost) W_2^T \odot (1 - A_1^2)$

This is the Hadamard product (element wise product) not a matrix multiplication

Compute $\frac{\partial}{\partial W_1} cost = \frac{1}{m} X^T \frac{\partial}{\partial Z_1} cost$

Compute $\frac{\partial}{\partial \beta_1} cost = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial Z_1} cost$

Neural Networks

$\beta_2, \beta_1, \beta_2$

partial derivative of the cost

ameters β_2, W_2, β_1, W_1...

$\frac{\partial}{\partial \beta_2} cost$ $\frac{\partial}{\partial W_2} cost$ $\frac{\partial}{\partial \beta_1} cost$ $\frac{\partial}{\partial W_1} cost$

p 4: Compute step sizes...

$\times learning\_rate$

$\times learning\_rate$

values for $\beta_1, W_1, \beta_2, W_2$

$W_2 = W_2 - step\_size_{W_2}$

$W_1 = W_1 - step\_size_{W_1}$

nd repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$\beta_2 = \beta_2 - step\_size_{\beta_2}$ $W_2 = W_2 - step\_size_{W_2}$

$\beta_1 = \beta_1 - step\_size_{\beta_1}$ $W_1 = W_1 - step\_size_{W_1}$

Step 6: Go to step 2 and repeat

Update Parameters

```go
// UpdateParams updates network parameters using gradient descent.
// Update rule: parameter = parameter - learning_rate * gradient
//
// Parameters:
//   - params: network parameters to update (W1, B1, W2, B2)
//   - gradients: computed gradients (DW1, DB1, DW2, DB2)
//   - learningRate: step size for gradient descent
func UpdateParams(params *NNParams, gradients *Gradients, learningRate float64) {
    w1Rows, w1Cols := params.W1.Dims()
    w2Rows, w2Cols := params.W2.Dims()
    b1Rows, _ := params.B1.Dims()
    b2Rows, _ := params.B2.Dims()

    // W1 = W1 - learning_rate * dW1
    for i := range w1Rows {
        for j := range w1Cols {
            params.W1.Set(i, j, params.W1.At(i, j)-learningRate*gradients.DW1.At(i, j))
        }
    }

    // B1 = B1 - learning_rate * dB1
    for i := range b1Rows {
        params.B1.Set(i, 0, params.B1.At(i, 0)-learningRate*gradients.DB1.At(i, 0))
    }

    // W2 = W2 - learning_rate * dW2
    for i := range w2Rows {
        for j := range w2Cols {
            params.W2.Set(i, j, params.W2.At(i, j)-learningRate*gradients.DW2.At(i, j))
        }
    }

    // B2 = B2 - learning_rate * dB2
    for i := range b2Rows {
        params.B2.Set(i, 0, params.B2.At(i, 0)-learningRate*gradients.DB2.At(i, 0))
    }
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1$ $A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Feedforward Neural Network

```go
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
type FeedforwardNN struct {
    n0           int
    n1           int
    n2           int
    learningRate float64
    iterations   int
    params       *NNParams
    cache        *Cache
    grads        *Gradients
}

// NewFeedforwardNN creates a new feedforward neural network.
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
func NewFeedforwardNN(n0, n1, n2 int, learningRate float64, iterations int) *FeedforwardNN {
    ffn := &FeedforwardNN{
        n0:           n0,
        n1:           n1,
        n2:           n2,
        learningRate: learningRate,
        iterations:   iterations,
        params:       &NNParams{},
        cache:        &Cache{},
        grads:        &Gradients{},
    }

    // Initialize parameters
    ffn.params.Initialize(n0, n1, n2)

    // Pre-allocate cache matrices (for batch size m, we'll allocate during train)
    // Cache and gradients will be allocated based on actual batch size in Train

    return ffn
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Feedforward Neural Network

```go
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
type FeedforwardNN struct {
    n0           int
    n1           int
    n2           int
    learningRate float64
    iterations   int
    params       *NNParams
    cache        *Cache
    grads        *Gradients
}

// NewFeedforwardNN creates a new feedforward neural network.
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
func NewFeedforwardNN(n0, n1, n2 int, learningRate float64, iterations int) *FeedforwardNN {
    ffn := &FeedforwardNN{
        n0:           n0,
        n1:           n1,
        n2:           n2,
        learningRate: learningRate,
        iterations:   iterations,
        params:       &NNParams{},
        cache:        &Cache{},
        grads:        &Gradients{},
    }

    // Initialize parameters
    ffn.params.Initialize(n0, n1, n2)

    // Pre-allocate cache matrices (for batch size m, we'll allocate during train)
    // Cache and gradients will be allocated based on actual batch size in Train

    return ffn
}
```

Struct representing the Feedforward Neural Network

n_0 = Number of neurons in the input layer (L_0)

n_1 = Number of neurons in the hidden layer ($L1$)

n_2 = Number of neurons in the output layer (L_2)

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Feedforward Neural Network

```go
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
type FeedforwardNN struct {
    n0           int
    n1           int
    n2           int
    learningRate float64
    iterations   int
    params       *NNParams
    cache        *Cache
    grads        *Gradients
}

// NewFeedforwardNN creates a new feedforward neural network.
// Parameters:
//    - n0: number of input features
//    - n1: number of hidden layer neurons
//    - n2: number of output classes
//    - learningRate: learning rate for gradient descent
//    - iterations: number of training iterations
func NewFeedforwardNN(n0, n1, n2 int, learningRate float64, iterations int) *FeedforwardNN {
    ffn := &FeedforwardNN{
        n0:           n0,
        n1:           n1,
        n2:           n2,
        learningRate: learningRate,
        iterations:   iterations,
        params:       &NNParams{},
        cache:        &Cache{},
        grads:        &Gradients{},
    }

    // Initialize parameters
    ffn.params.Initialize(n0, n1, n2)

    // Pre-allocate cache matrices (for batch size m, we'll allocate during train)
    // Cache and gradients will be allocated based on actual batch size in Train

    return ffn
}
```

Struct representing the Feedforward Neural Network

n_0 = Number of neurons in the input layer (L_0)

n_1 = Number of neurons in the hidden layer ($L1$)

n_2 = Number of neurons in the output layer (L_2)

Constructor initializes the params ($W_1, \beta_1, W_2, \beta_2$)

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Gradient Descent

Step 1: Start with initial values for $W_1, W_2, \beta_1, \beta_2$

Step 2: Compute… $Z_1 = XW_1 + \beta_1 \quad A_1 = tanh(Z_1)$

$$Z_2 = A_1 W_2 + \beta_2$$

$$\hat{Y} = softmax(Z_2)$$

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$…

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes…

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

Step 3: Compute the partial derivative of the cost function w.r.t the parameters β_2, W_2, β_1, W_1...

$$\frac{\partial}{\partial \beta_2} cost \quad \frac{\partial}{\partial W_2} cost \quad \frac{\partial}{\partial \beta_1} cost \quad \frac{\partial}{\partial W_1} cost$$

Step 4: Compute step sizes...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for β_1,W_1, β_2, W_2

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \quad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \quad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Forward Propagation

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W_2} = \frac{\partial}{\partial W_2} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Forward Propagation

Compute Cost (for display & debugging only)

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W} = \frac{\partial}{\partial W} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Forward Propagation

Compute Cost (for display & debugging only)

Back Propagation

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W} = \frac{\partial}{\partial W} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

Step 5: Compute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Forward Propagation

Compute Cost (for display & debugging only)

Back Propagation

Update Parameters

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W} = \frac{\partial}{\partial W} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

...pute new values for $\beta_1, W_1, \beta_2, W_2$

$$\beta_2 = \beta_2 - step\_size_{\beta_2} \qquad W_2 = W_2 - step\_size_{W_2}$$

$$\beta_1 = \beta_1 - step\_size_{\beta_1} \qquad W_1 = W_1 - step\_size_{W_1}$$

Step 6: Go to step 2 and repeat

Train() Method

```go
// Train trains the neural network using the provided training data.
// Algorithm:
//  1. Initialize parameters
//  2. For each iteration:
//     - Forward propagation
//     - Compute cost
//     - Backward propagation
//     - Update parameters
//     - Print cost every 10% of iterations
//
// Parameters:
//   - x: training input data (m × n0)
//   - y: training labels (m × n2), one-hot encoded
//
// Returns:
//   - costHistory: slice of cost values recorded during training
func (ffn *FeedforwardNN) Train(x *mat.Dense, y *mat.Dense) []float64 {
    m, _ := x.Dims()

    // Allocate cache matrices based on batch size
    ffn.cache.Z1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.A1 = mat.NewDense(m, ffn.n1, nil)
    ffn.cache.Z2 = mat.NewDense(m, ffn.n2, nil)
    ffn.cache.Yhat = mat.NewDense(m, ffn.n2, nil)

    // Allocate gradient matrices
    ffn.grads.DW1 = mat.NewDense(ffn.n0, ffn.n1, nil)
    ffn.grads.DB1 = mat.NewDense(ffn.n1, 1, nil)
    ffn.grads.DW2 = mat.NewDense(ffn.n1, ffn.n2, nil)
    ffn.grads.DB2 = mat.NewDense(ffn.n2, 1, nil)

    // Track cost history
    costHistory := make([]float64, 0, ffn.iterations)

    // Training loop
    for i := range ffn.iterations {
        // Forward propagation
        ForwardPropagation(x, ffn.params, ffn.cache)

        // Compute cost
        cost := CostFunction(ffn.cache.Yhat, y)
        costHistory = append(costHistory, cost)

        // Backward propagation
        BackPropagation(x, y, ffn.params, ffn.cache, ffn.grads)

        // Update parameters
        UpdateParams(ffn.params, ffn.grads, ffn.learningRate)

        // Print cost every 10% of iterations
        if i%(ffn.iterations/10) == 0 {
            println("Cost after iteration", i, ":", cost)
        }
    }

    return costHistory
}
```

Neural Networks

Allocate the cache matrices and gradient matrices. This ensures that we allocate memory upfront and reuse it rather than reallocating on every step

History of the Cost (for display and debugging only). Not required for the training or inference

Forward Propagation

Compute Cost (for display & debugging only)

Back Propagation

Update Parameters

Gradient Descent iterates *iteration* times

Step 3: Compute the partial derivative of the cost function w.r.t the parameters $\beta_2, W_2, \beta_1, W_1$...

$$step\_size_{\beta_2} = \frac{\partial}{\partial \beta_2} cost \times learning\_rate$$

$$step\_size_{W} = \frac{\partial}{\partial W} cost \times learning\_rate$$

$$step\_size_{\beta_1} = \frac{\partial}{\partial \beta_1} cost \times learning\_rate$$

$$step\_size_{W_1} = \frac{\partial}{\partial W_1} cost \times learning\_rate$$

...pute new values for $\beta_1, W_1, \beta_2, W_2$

Step 6: Go to step 2 and repeat

Predict() Method


```go
// Predict makes predictions on new data using the trained network.
// Performs forward propagation to compute output probabilities.
//
// Parameters:
//   - x: input data (m × n0), where m = number of samples
//
// Returns:
//   - predictions: output probabilities (m × n2) from softmax layer
func (ffn *FeedforwardNN) Predict(x *mat.Dense) *mat.Dense {
    m, _ := x.Dims()

    // Create temporary cache for prediction (don't reuse training cache)
    predCache := &Cache{
        Z1:   mat.NewDense(m, ffn.n1, nil),
        A1:   mat.NewDense(m, ffn.n1, nil),
        Z2:   mat.NewDense(m, ffn.n2, nil),
        Yhat: mat.NewDense(m, ffn.n2, nil),
    }

    // Forward propagation
    ForwardPropagation(x, ffn.params, predCache)

    // Return predictions (probabilities for each class)
    return predCache.Yhat
}
```

Complete Code with Training Data:
https://github.com/arrsingh/ai-tutorials-go/blob/mainline/tutorials/snn.go

Predict() Method


```go
// Predict makes predictions on new data using the trained network.
// Performs forward propagation to compute output probabilities.
//
// Parameters:
//   - x: input data (m × n0), where m = number of samples
//
// Returns:
//   - predictions: output probabilities (m × n2) from softmax layer
func (ffn *FeedforwardNN) Predict(x *mat.Dense) *mat.Dense {
    m, _ := x.Dims()

    // Create temporary cache for prediction (don't reuse training cache)
    predCache := &Cache{
        Z1:   mat.NewDense(m, ffn.n1, nil),
        A1:   mat.NewDense(m, ffn.n1, nil),
        Z2:   mat.NewDense(m, ffn.n2, nil),
        Yhat: mat.NewDense(m, ffn.n2, nil),
    }

    // Forward propagation
    ForwardPropagation(x, ffn.params, predCache)

    // Return predictions (probabilities for each class)
    return predCache.Yhat
}
```

During inference we only calculate the predicted value $\hat{Y}$ using forward propagation

Complete Code with Training Data:
https://github.com/arrsingh/ai-tutorials-go/blob/mainline/tutorials/snn.go

Related Tutorials & Textbooks

Neural Networks

An introduction to Neural Networks starting from a foundation of linear regression, logistic classification and multi class classification models along with the matrix representation of a neural network generalized to *l* layers with *n* neurons

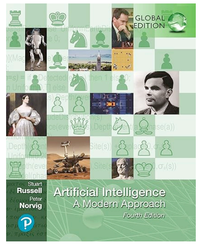
Forward and Back Propagation in Neural Networks

A deep dive into how Neural Networks are trained using Gradient Descent. Output predictions, are compared to observations to calculate loss and Backward propagation then computes gradients by working backward through the network

Gradient Descent for Multiple Regression

Gradient Descent algorithm for multiple regression and how it can be used to optimize k + 1 parameters for a Linear model in multiple dimensions.

Recommended Textbooks



Artificial Intelligence: A Modern Approach
by Peter Norvig, Stuart Russell

For a complete list of tutorials see:
https://arrsingh.com/ai-tutorials