

# Deep Belief Networks

Umar Arshad      @arshad\_umar  
Arrayfire      @arrayfire

# Connection Instructions

- Navigate to [nvlabs.qwiklab.com](https://nvlabs.qwiklab.com)
- Login or create a new account
- Select the “Instructor-Led Hands-on Labs” class
- Find the lab called “<INSERT LAB NAME>” and click Start
- After a short wait, lab instance connection information will be shown
- Please ask Lab Assistants for help!

# ArrayFire

- CUDA and OpenCL experts
  - since 2007
- Headquartered in Atlanta, GA
- In search for the best and the brightest
  - Engineers from around the world
- Expert domain experience in a wide variety of fields
  - Computer Vision
  - Machine Learning
  - Financial
  - etc.

# ArrayFire Consulting Services

- Custom software development services
- Deep experience working with thousands of customers
  - Analysis
  - Acceleration
  - Algorithm development
- Expert software engineers
  - Large scale software development experience
  - Production quality code
  - Extensive domain knowledge



# ArrayFire Training

- 2-4 Day CUDA or OpenCL training
  - On site or at our headquarters
- Taught by one or two performance engineer
  - We have seen it all and know how to fix things
- Hands on labs
  - You will not be copying code
  - Run on GPU hardware
- Customized for your application
  - Examples target your use-case
- Only C/C++ experience required

# ArrayFire the Library

- A general purpose computational library
- Backends for CUDA, OpenCL and CPU
- Cross Platform
- Open Source (BSD - 3 - clause)
- Concentrate on performance and ease of use
- JIT
- Hundreds of Functions

# Machine Learning

- Excellent for modeling highly dimensional data
- Pattern recognition
- Decision making
- Meta-programming

# MNIST Dataset

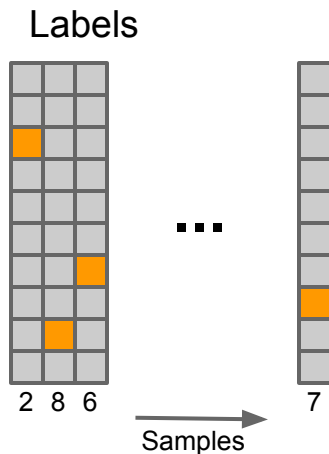
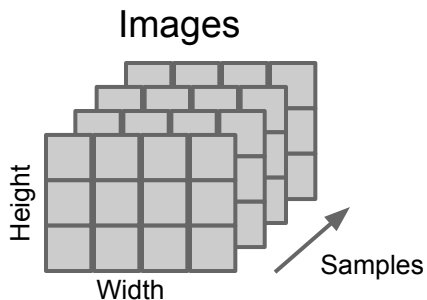
- Dataset of handwritten digits
- 60,000 samples from ~250 writers
- 28x28 grayscale pixels



Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998

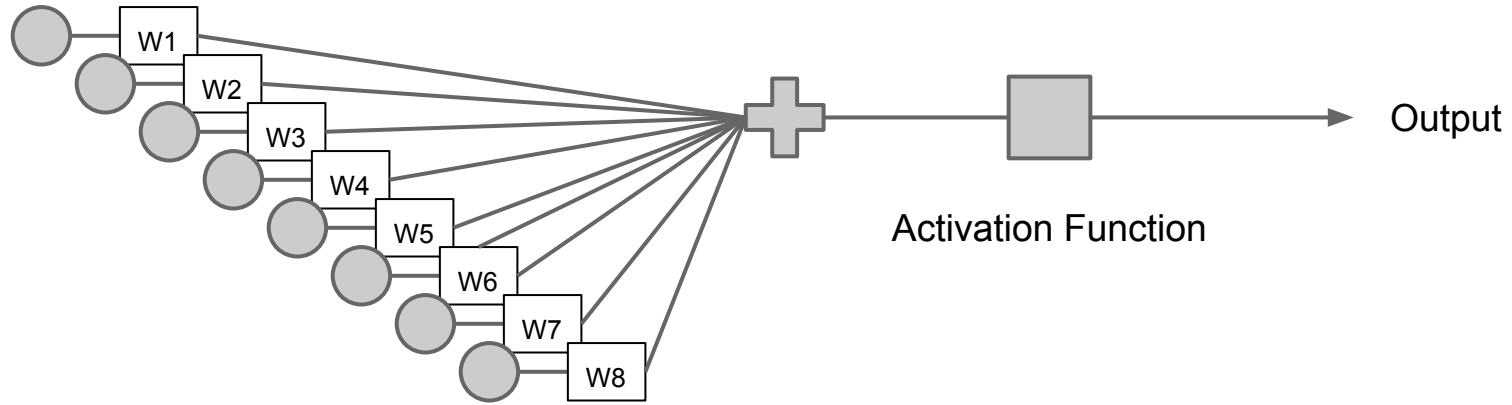
# Loading MNIST

```
array train_images, train_targets;  
array test_images, test_targets;  
int num_train, num_test, num_classes;  
  
// Load mnist data  
setup_mnist<true>(&num_classes, &num_train, &num_test,  
                 train_images, test_images,  
                 train_targets, test_targets, 1.0);
```



# Perceptron

- Introduced in the late 1950's
- Linear classifier



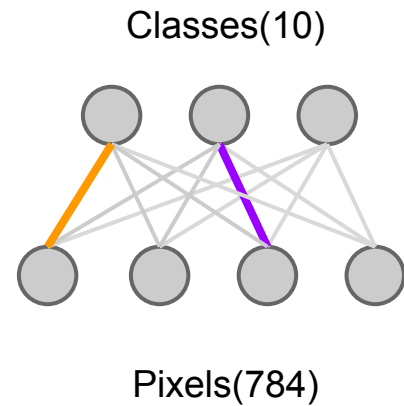
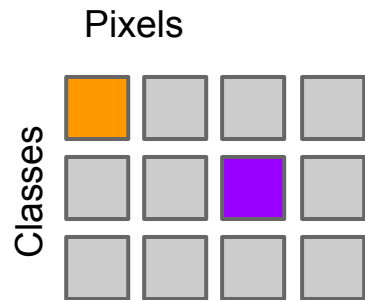


# Teaching a Perceptron

- Initialize weights to zero
- Generate response
- Calculate error
- Update weights
- Repeat

# Perceptron

## Weights



# ArrayFire API

- Creating arrays in ArrayFire

```
array zeros    = constant(0, 5);    // 0  
                                         // 0  
                                         // 0  
                                         // 0  
                                         // 0
```

```
array zeros2D = constant(0, 2, 3);  // 0, 0, 0  
                                         // 0, 0, 0
```

# Perceptron

```
//Initialize weights to 0
const int pixel_count = 28*28; //train_feat.dims(1);
const int num_labels = 10;      //train_targest.dims(1);
array weights = constant(0, pixel_count, num_labels);
```

# Teaching a Perceptron

- Initialize weights to zero
- Generate response
- Calculate error
- Update weights
- Repeat

# Response

- Sum of the input multiplied by the weights

$$response(p) = \sum w_i p_i$$

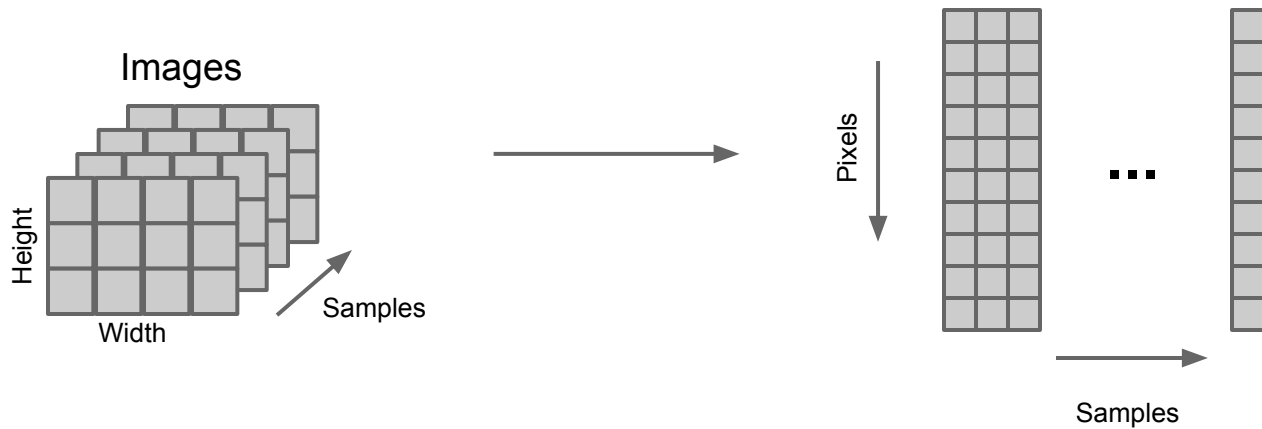
- Send result into an activation function

$$S(t) = \frac{1}{1 + e^{-t}}$$



# Response

- Flatten Images



# ArrayFire API

- Reshaping volume into a matrix

```
// Reshape images into feature vectors
```

```
array out = moddims(in, dim0, dim1, dim2, dim3);
```

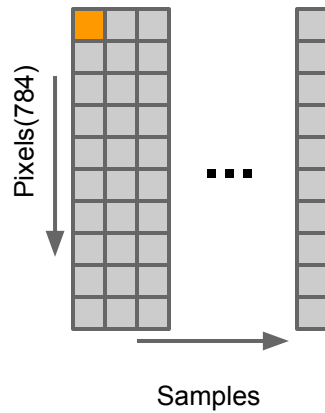
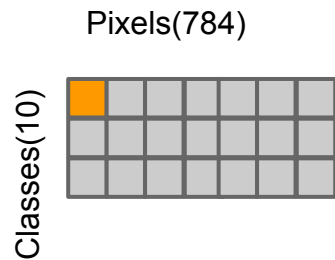
```
// Reshape images into feature vectors
```

```
array train_feats = moddims(train_images, pixel_count, num_train);
```

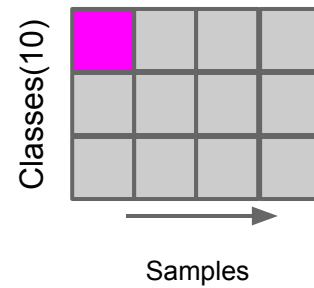
# Response

```
// Reshape images into feature vectors  
array train_feats = moddims(train_images, pixel_count, num_train);  
array test_feats  = moddims(test_images , pixel_count, num_test );
```

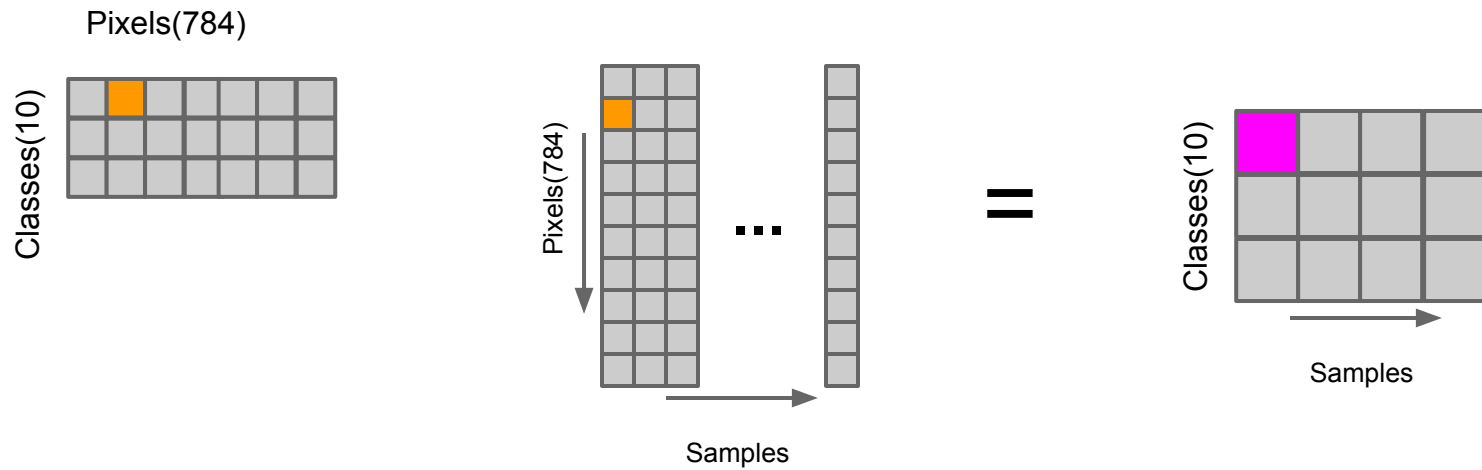
# Response



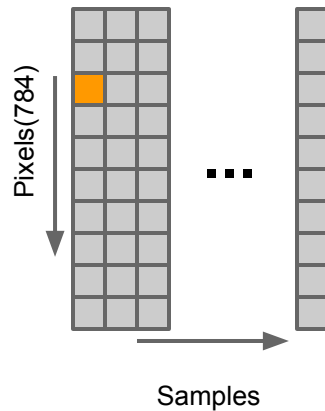
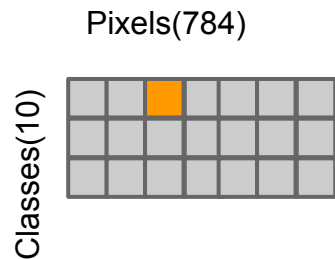
=



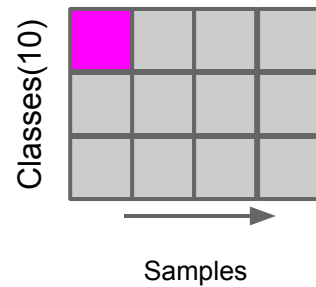
# Response



# Response



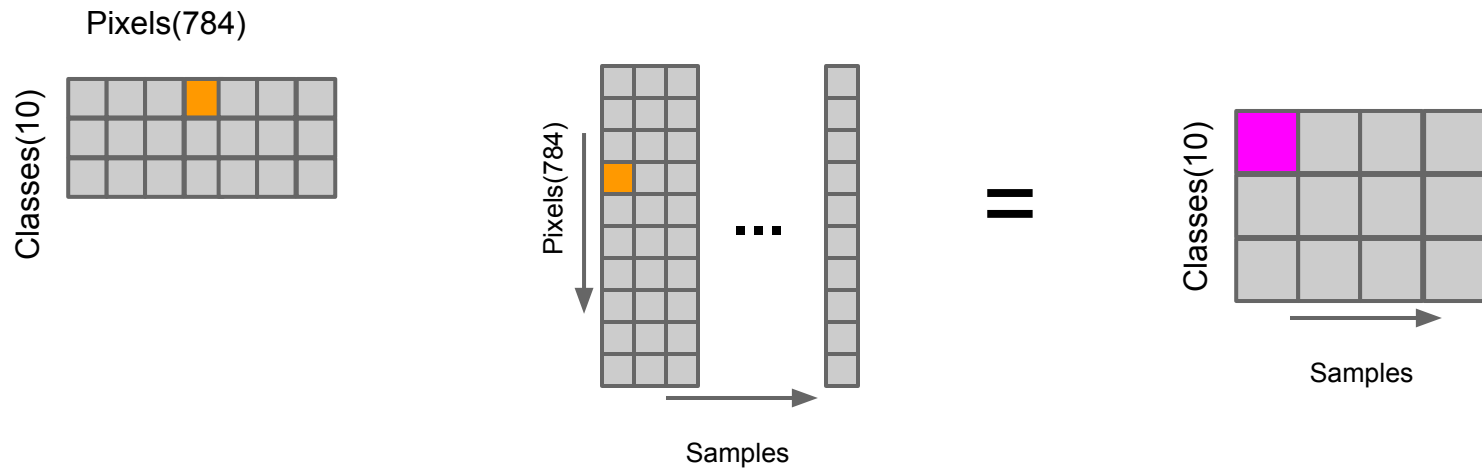
||





# Response

Matrix Multiply!



# ArrayFire API

```
array response = matmul(weights, train_feats);
```

# Activation Function

- Sigmoid Function

$$S(t) = \frac{1}{1 + e^{-t}}$$

```
// Activation function  
array sigmoid(const array &val)  
{  
    return 1 / (1 + exp(-val));  
}
```

# Prediction

```
array prediction = sigmoid(matmul(weights, train_feats));
```

# Teaching a Perceptron

- Initialize weights to zero
- Generate response
- Update weights
- Repeat

# Calculating Error

- Subtract the expected output with prediction
- Multiply with the learning rate

$$\Delta w = \alpha(\textit{expected}_p - \textit{prediction}_p) * \textit{pixel}_i$$

```
array err = train_targets - prediction;  
weights += learning_rate * (matmulNT(err, train_feats));
```



# Teaching a Perceptron

- Initialize weights to zero
- Generate response
- Update weights
- Repeat

# Repeat

```
for(int i = 0; i < 100; i++)
{
    array prediction = sigmoid(matmul(weights, train_feats));
    array err = train_targets - prediction;

    float mean_abs_error = mean<float>(abs(err));
    printf("err: %0.4f\n", mean_abs_error);

    weights += learning_rate * (matmulNT(err, train_feats));
}
```

# Results

## Measure Accuracy

```
float accuracy(const array& predicted, const array& target)
{
    array val, plabels, tlabels;
    max(val, tlabels, target, 0);
    max(val, plabels, predicted, 0);
    return 100 * count<float>(plabels == tlabels) / tlabels.elements();
}
```

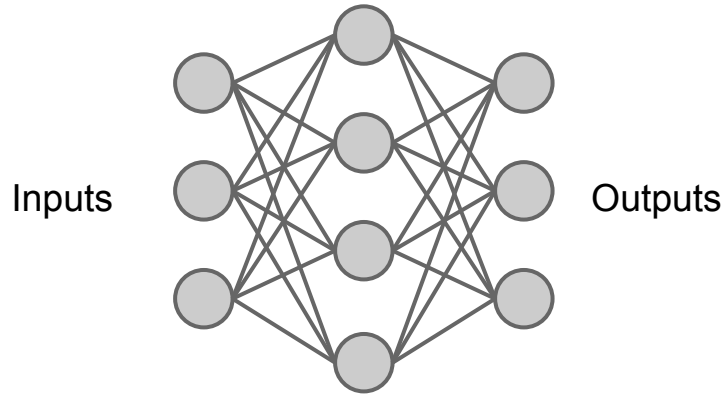
82.03%

# Perceptron

- Improvements
  - Smaller batches
  - Variable learning rate
- Linear Classifier!
  - Handwritten digit recognition cannot be solved by a linear classifier
- Additional layers are required

# Neural Networks

- Made up of one or more layers of neurons
- Hidden layers updated using back propagation



# Back Propagation

- Hidden layers do not have an expected output
- Calculating error on output
- Send in data from the output layer back into network
- Gradient descent

# Back Propagation

```
void ann::back_propagate(const vector<array> signal, const array &target, const double &alpha){
    // Get error for output layer
    array out = signal[num_layers - 1];
    array err = (out - target);
    int m = target.dims(0);
    for (int i = num_layers - 2; i >= 0; i--) {
        array in = add_bias(signal[i]);
        array delta = (deriv(out) * err).T();

        // Adjust weights
        array grad = -(alpha * matmul(delta, in)) / m;
        weights[i] += grad.T();

        // Input to current layer is output of previous
        out = signal[i];
        err = matmul(weights[i], delta).T();

        // Remove the error of bias and propagate backward
        err = err(span, seq(1, out.dims(1)));
    }
}
```

# Results

- Accuracy: 93.90%
- Time: 31.30 seconds
- Epoch: 250

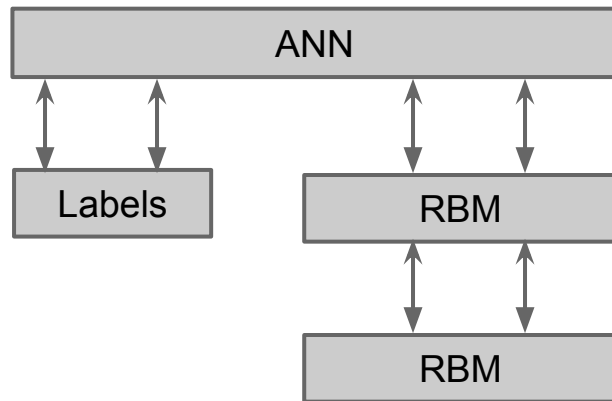


# Back Propagation

- Effective
- Slow for deeper networks
- Requires labeled data

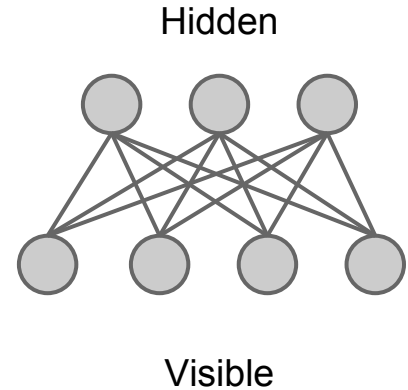
# Deep Belief Nets

- A neural network made of multiple layers Restricted Boltzmann Machines
- Deep belief networks are great with unlabeled data



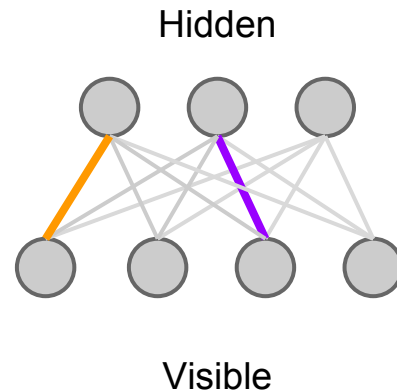
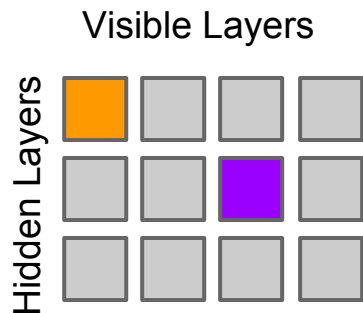
# Restricted Boltzmann Machine

- A neural network with one hidden layer
- Each hidden neuron is connected to every visible neuron
- The connection has a weight which represents how strongly the neuron reacts to that input
- A bias is associated with both hidden and visible neurons



# Restricted Boltzmann Machine

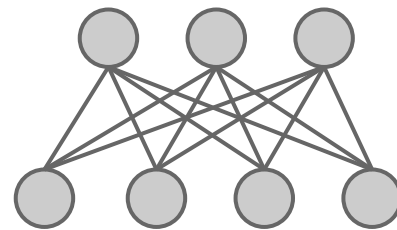
## Data Representation



# RBM

Lets create our RBM

```
class rbm
{
    array weights;
    array h_bias;
    array v_bias;
public:
    rbm(visible_size, hidden_size)
        : weights( constant(0, hidden_size, visible_size))
        , h_bias(   constant(0, 1, hidden_size))
        , v_bias(   constant(0, 1, visible_size))
    {}
};
```



Visible

# Training RBM

- Feed input into RBM
- Calculate the response
- Feed the response back through the network
- Calculate the error of the reconstruction
- Adjust the weights

# Building the DBN

- Feed the output of the previous layer to the next
- Learns higher level features
- Use back propagation to fine tune the data

# Results

- Accuracy: 93.46%
- Time: 13.27 seconds
- Epoch: 108



# Improvements

- More data
- Larger network
- Learning rate
- Longer iterations

# Questions