Out[5]: The raw code for this IPython notebook is by default hidden for easier reading. To toggle on/off the raw code, click here.

```
Using TensorFlow backend.
```

Notebook Structure

1) Introduction of Machine Learning	@developed by: Matteo Alberti @date: 12/2018
Core Modules	
2) Activation functions 3) Forward Propagation 3.1) Costs & Accuracy 4) Backpropagation 5) Optimizers 6) Learning Rate 7) Dropout & Regolarization	
8) Grid Search	

1) Introduction to Machine & Deep Learning

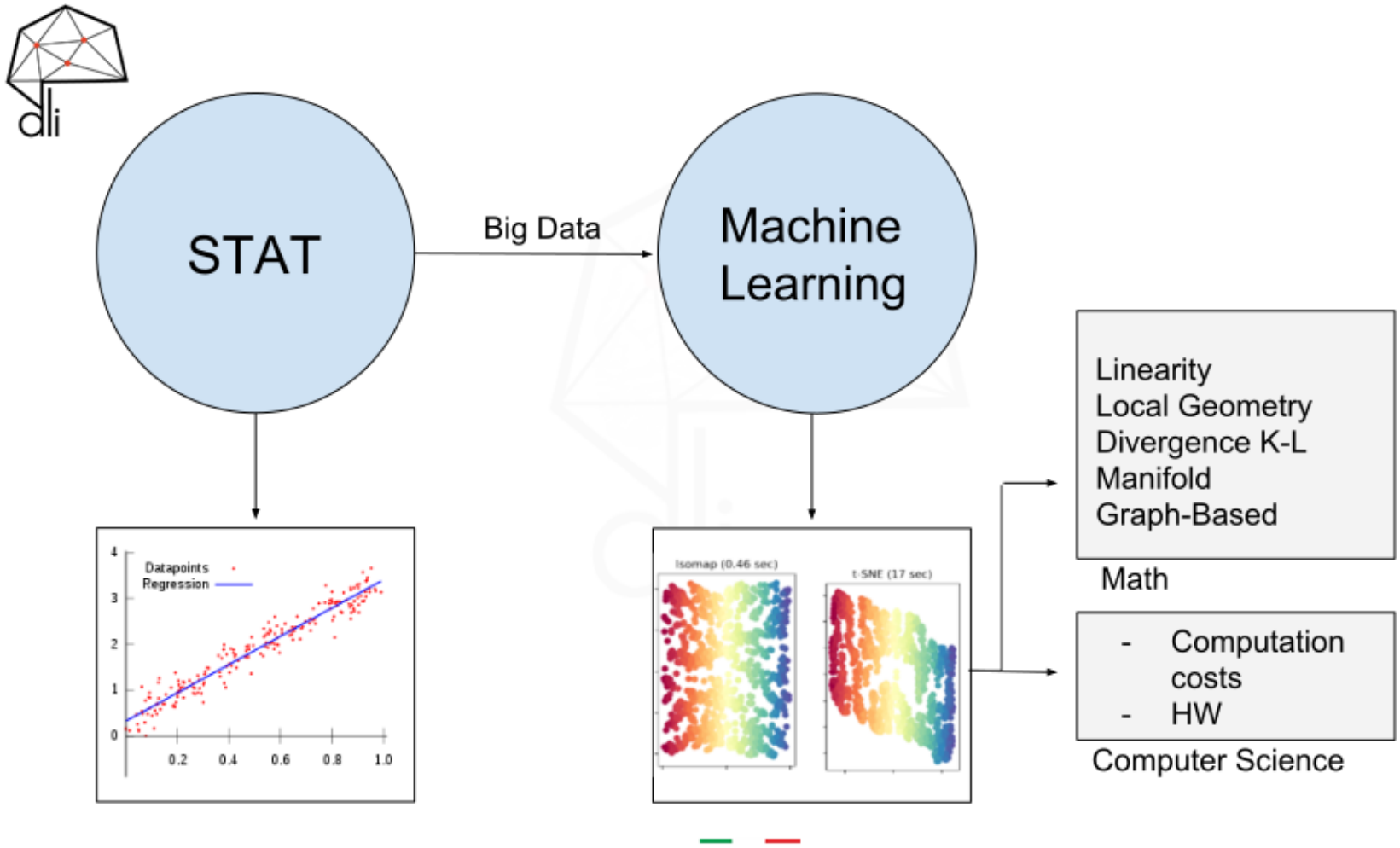


Figure 1. From Statistics to Machine Learning

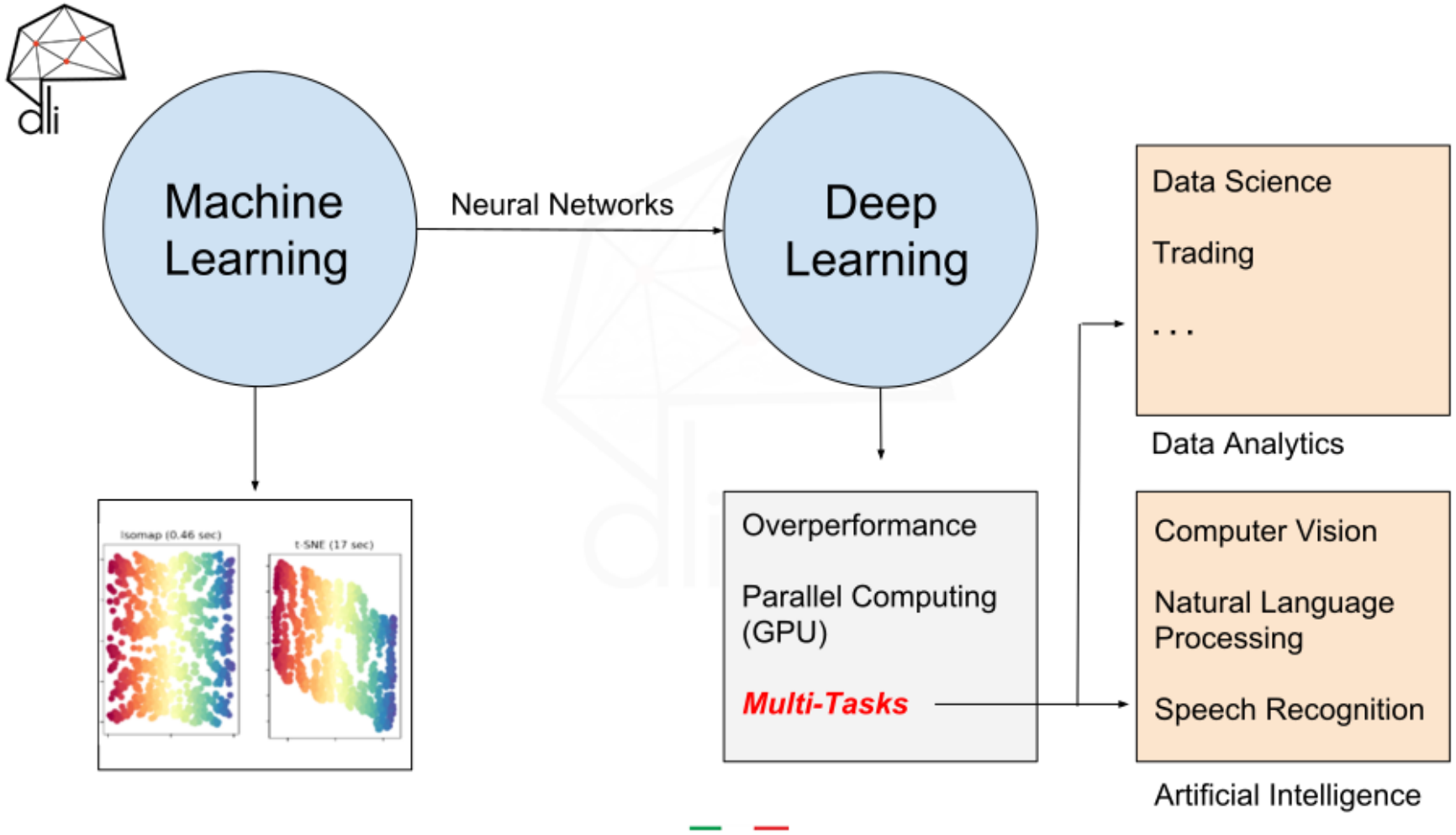


Figure 2. From Machine Learning to Deep Learning

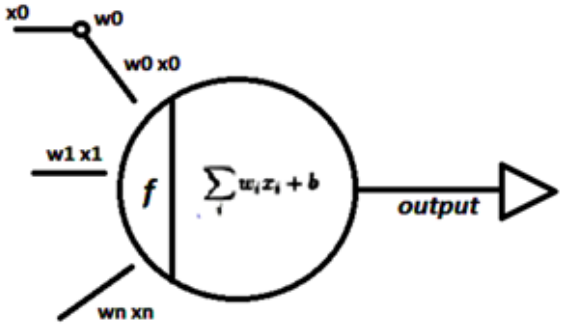
From Perceptron to Multi Layer Perceptron (MLP)

$a_i = f(\sum_i(w_i x_i) + b_i)$ "Not-Linear Discrimination"

Where $f(x)$ Activation Function
without $f(x)$ ANN regresses into linear form:

$$L_T = W_T\ L_{T-1} = W_T W_{T-1} L_{T-2} = \cdots (W_T W_{T-1} \ldots W_1) L_0 = W L_0$$

One single neuron isn't able to approximate enough well some decision functions (typically not linear).
We can model that stacking "layers" of neurons.



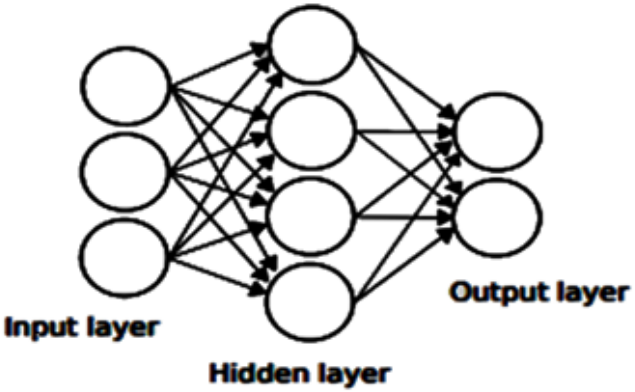


Figure 3. Perceptron to MLP

2) Activation Functions

The Nonlinear Activation Functions are mainly divided on the basis of their range

Sigmoid or Logistic Function

$$f(x) = \frac{1}{1 + e^{-x}} \qquad f'(x) = f(x) \cdot (1 - f(x))$$

It is especially used for models where we have to predict the probability as an output. Between 0 and 1

- Differentiable
- Monotonic but function's derivative is not

Note:

- The softmax function is a more generalized logistic activation function which is used for multiclass classification

Tanh

$$f(x) = \operatorname{thg} x = \frac{\sinh x}{\cosh x} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad f'(x) = 1 - f(x)^2$$

- The range of the tanh function is from (-1 to 1)
- negative inputs will be mapped strongly negative
- mainly used classification between two classes

ReLU

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq x \end{cases} \qquad f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq x \end{cases}$$

- Range: [0, inf)
- Deep Learning & Computer Vision Applications
- The function and its derivative both are monotonic

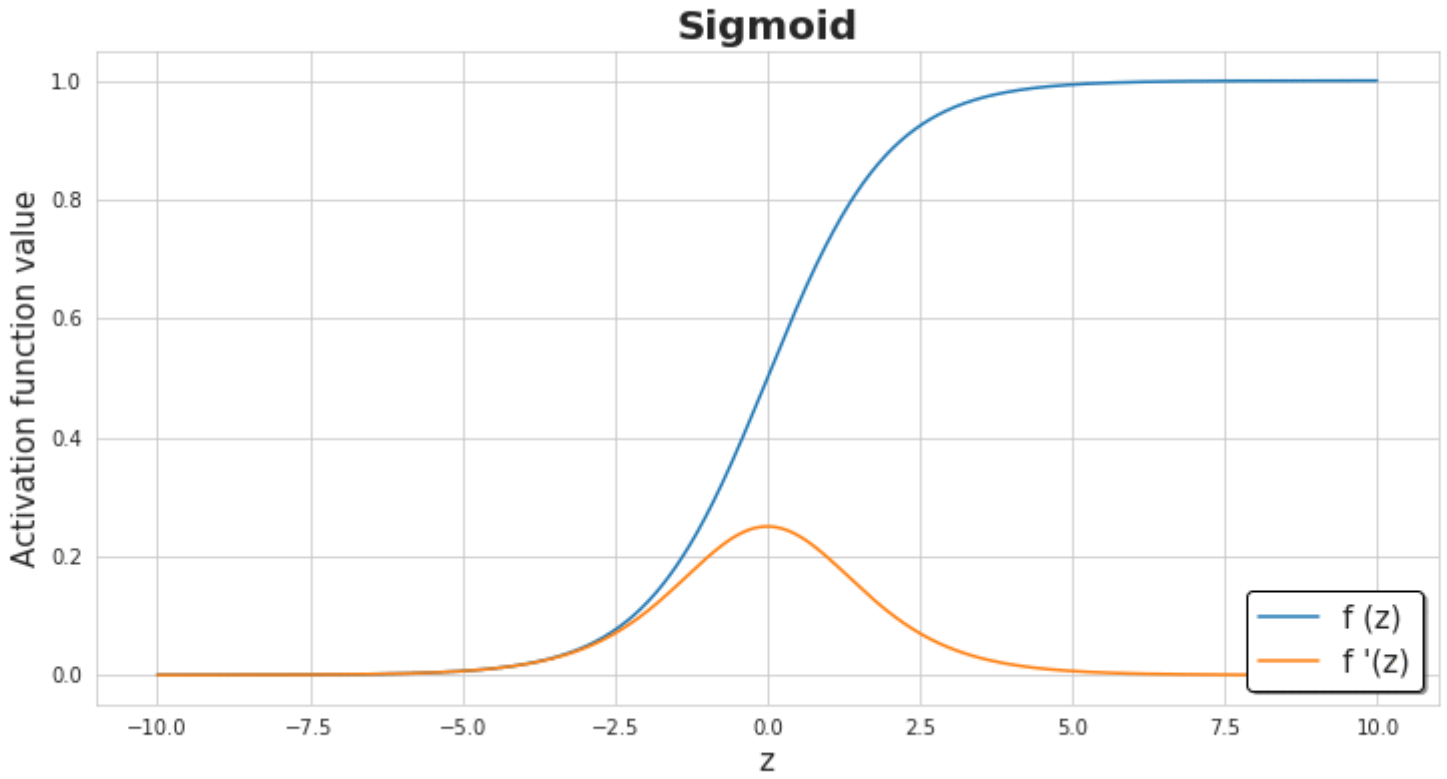
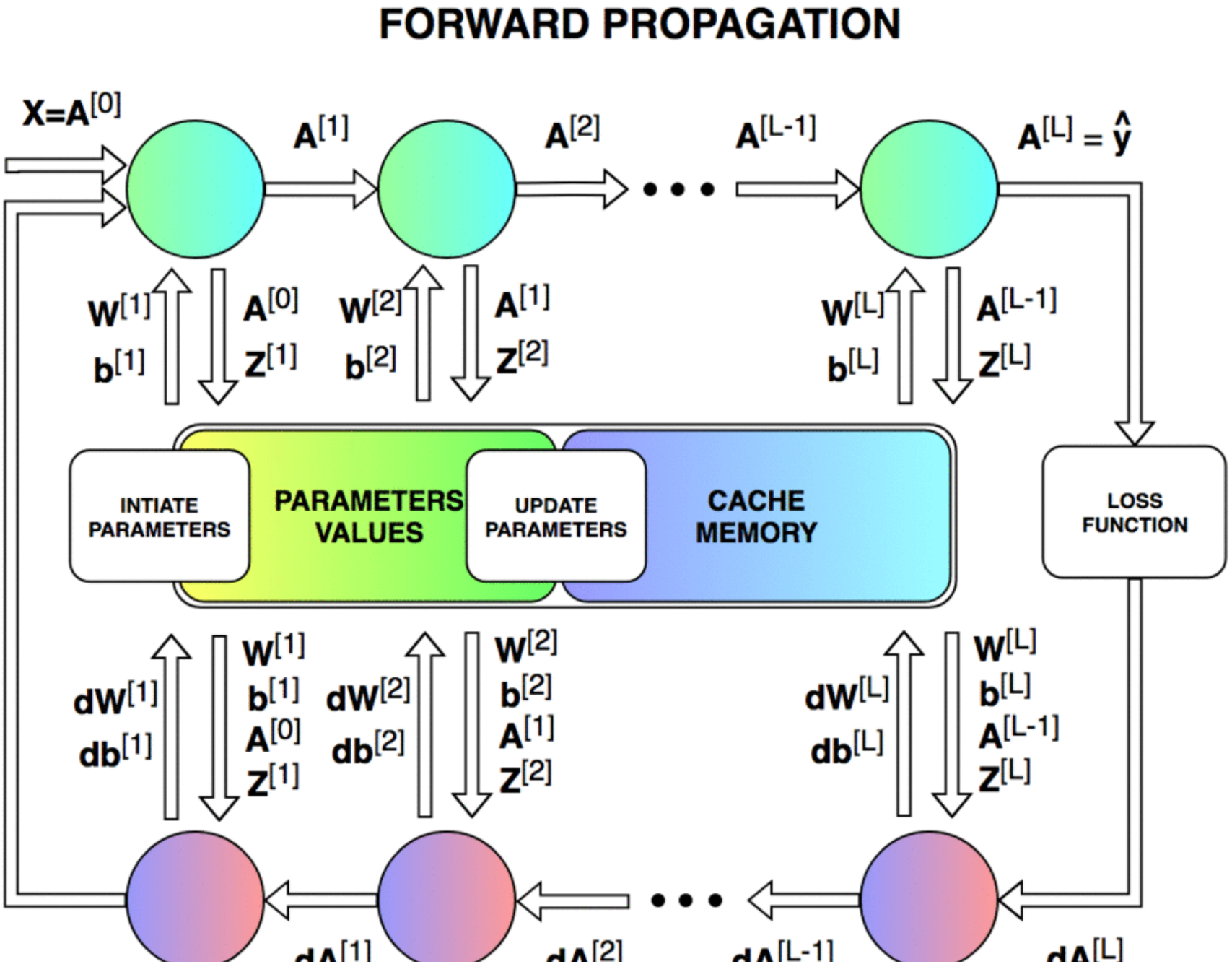


Figure 4. Activation functions used in the algorithm.

3) Forward & Backward Pass


FORWARD PROPAGATION
X=A[0]
A[1]
A[2]
A[L-1]
A[L] = ŷ
W[1]
b[1]
A[0]
Z[1]
W[2]
b[2]
A[1]
Z[2]
W[L]
b[L]
A[L-1]
Z[L]
INTIATE PARAMETERS
PARAMETERS VALUES
UPDATE PARAMETERS
CACHE MEMORY
LOSS FUNCTION
dW[1]
db[1]
W[1]
b[1]
A[0]
Z[1]
dW[2]
db[2]
W[2]
b[2]
A[1]
Z[2]
dW[L]
db[L]
W[L]
b[L]
A[L-1]
Z[L]
dA[1]
dA[2]
dA[L-1]
dA[L]

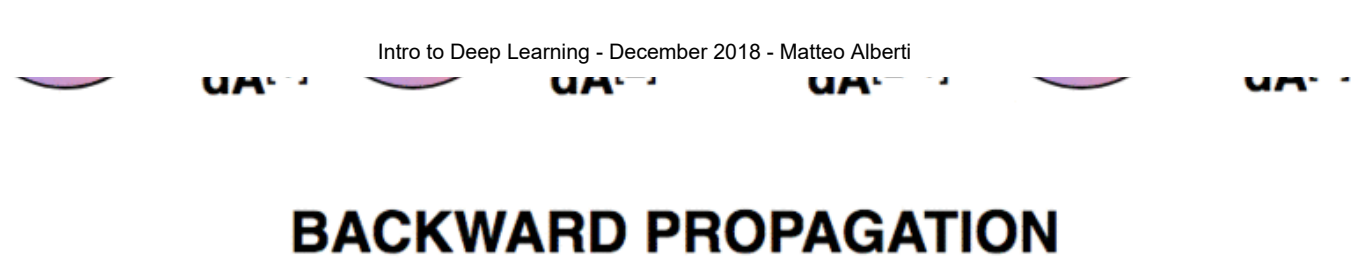


Figure 4.1 Visualization of Forward & Backward Pass

Forward Pass

$$y_i = f(\sum_i W_i x_i + b_i)$$

we're going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward though the network.

We figure out the total net input to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons.

Here's how we calculate the total net input for h_1:

$$\boldsymbol{Net}^{[l]} = \boldsymbol{W}^{[l]} \cdot \boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}$$

So if we have 2 entries

$$\boldsymbol{Net}^{[l]} = 0.15 \cdot 0.05 + 0.2 \cdot 0.1 + 0.35 \cdot 1 = 0.365$$

then squash it using the logistic function

$$Out_{[1]} = \frac{1}{1 + e^{-0.365}} = 0.612$$

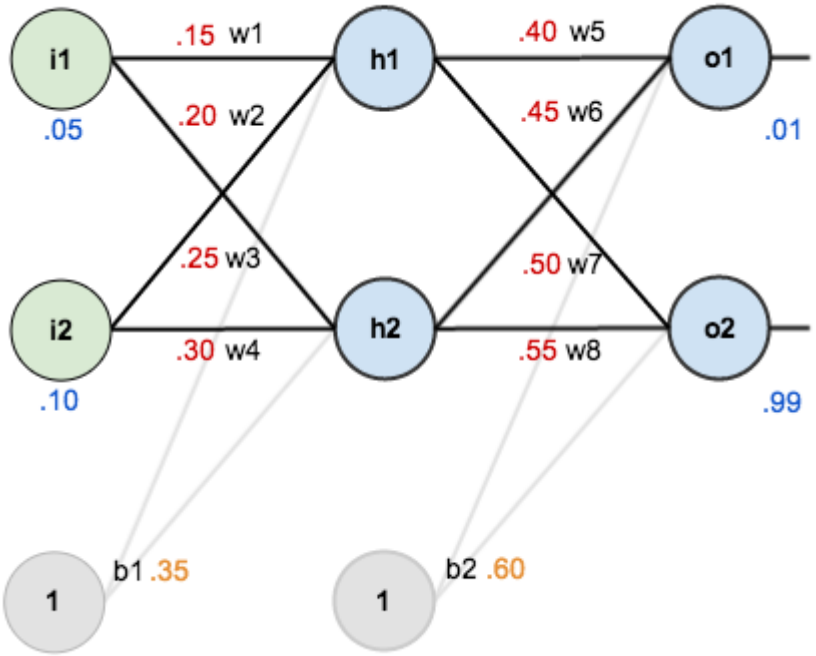


Figure 4.2 Forward Pass 2

Calculating Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error

Error 1

$$E_{tot} = \sum \frac{1}{2}(target - output)^2$$

So:

$$E_{tot} = \sum \frac{1}{2}(target - output)^2 = \frac{1}{2}(0.01 - 0.612)^2 = 0.24$$

Total Error

$$E_{tot} = \sum_{i=1}^{t} E_i = 0.28$$

Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

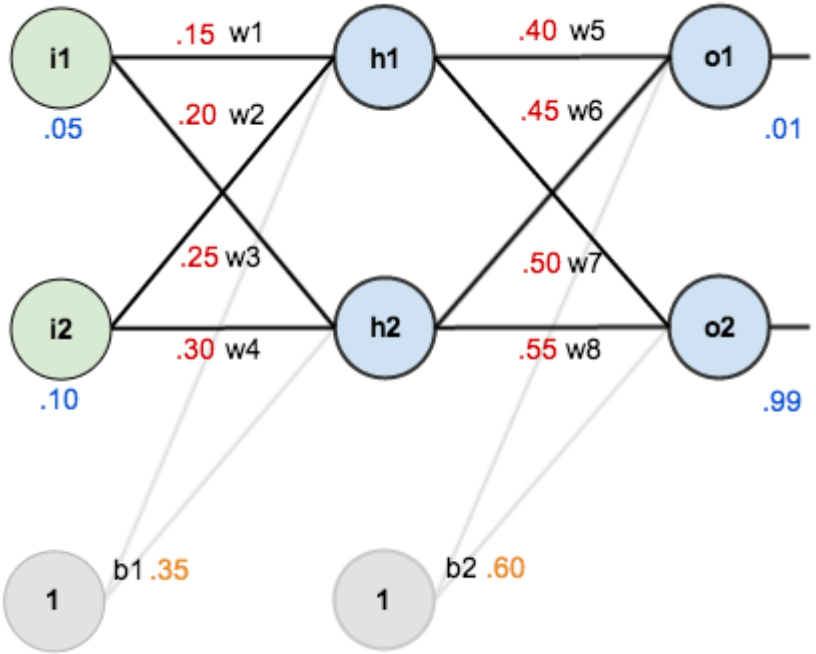


Figure 4.3 Throught Backward Pass

Consider w5. We want to know how much a change in w5 affects the total error (the partial derivative of E_total with respect to w5)

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial Net^1} * \frac{\partial Net^1}{\partial w_5} = 0.0754$$

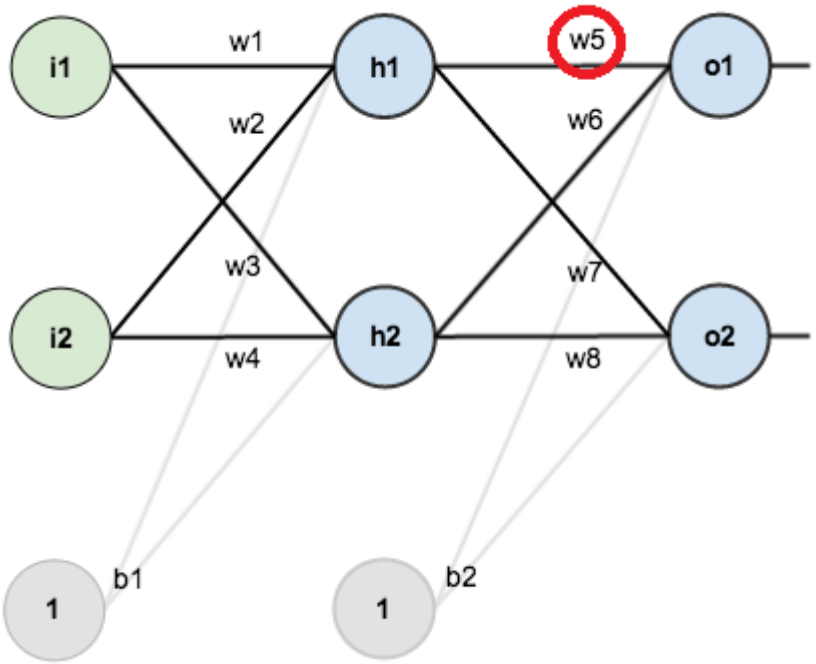


Figure 4.4 Throught Backward Pass 2

When we take the partial derivative of the total error with respect to out{o1}, the quantity

$$\frac{1}{2}(target_{o2} - out_{o2})^2$$

becomes zero because o1 does not affect it which means we're taking the derivative of a constant which is zero.

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5)

$$w_5^+ = w_5 - \mu * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.0754 = 0.392$$

We can repeat this process to get the new weights than we perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons

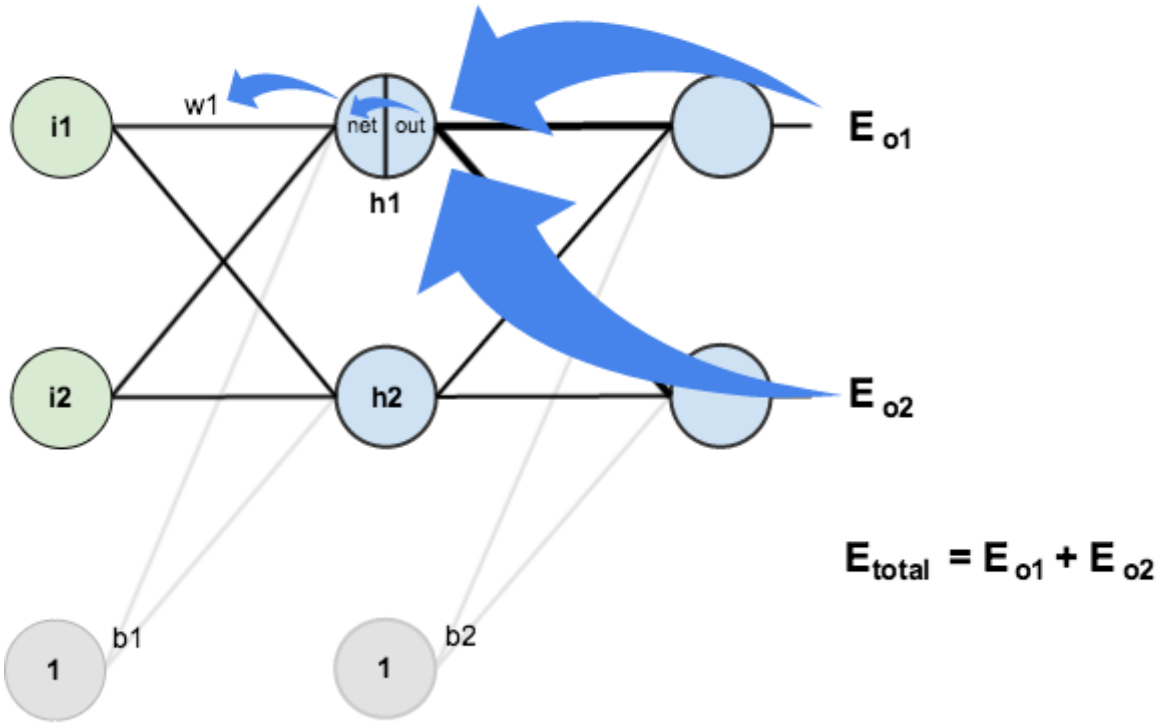


Figure 4.5 Throught Backward Pass 3

We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out1 affects both out1 and out2 therefore

$$\frac{\partial E_{total}}{\partial out_{h1}}$$

needs to take into consideration its effect on the both output neurons

Summary

1) **Calculate the forward phase** for each input-output pair $(\overrightarrow{x_d}, y_d)$ and store the results $\hat{y_d}$, a_j^k, and o_j^k for each node j in layer k by proceeding from layer 0, the input layer, to layer m, the output layer.

2) **Calculate the backward phase** for each input-output pair $(\overrightarrow{x_d}, y_d)$ and store the results $\frac{\partial E_d}{\partial w_{ij}^k}$ for each weight w_{ij}^k connecting node i in layer $k-1$ to node j in layer k by proceeding from layer m, the output layer, to layer 1, the input layer.

a) Evaluate the error term for the final layer δ_1^m by using the second equation.
b) Backpropagate the error terms for the hidden layers δ_j^k, working backwards from the final hidden layer $k = m - 1$, by repeatedly using the third equation.
c) Evaluate the partial derivatives of the individual error E_d with respect to w_{ij}^k by using the first equation.

3) **Combine the individual gradients** for each input-output pair $\frac{\partial E_d}{\partial w_{ij}^k}$ to get the total gradient $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$ for the entire set of input-output pairs $X = \{(\overrightarrow{x_1}, y_1), \ldots, (\overrightarrow{x_N}, y_N)\}$ by using the fourth equation (a simple average of the individual gradients).

4) **Update the weights** according to the learning rate α and total gradient $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$ by using the fifth equation (moving in the direction of the negative gradient).

Figure 4.6 BackPropagation

Visualize Gradient Descent

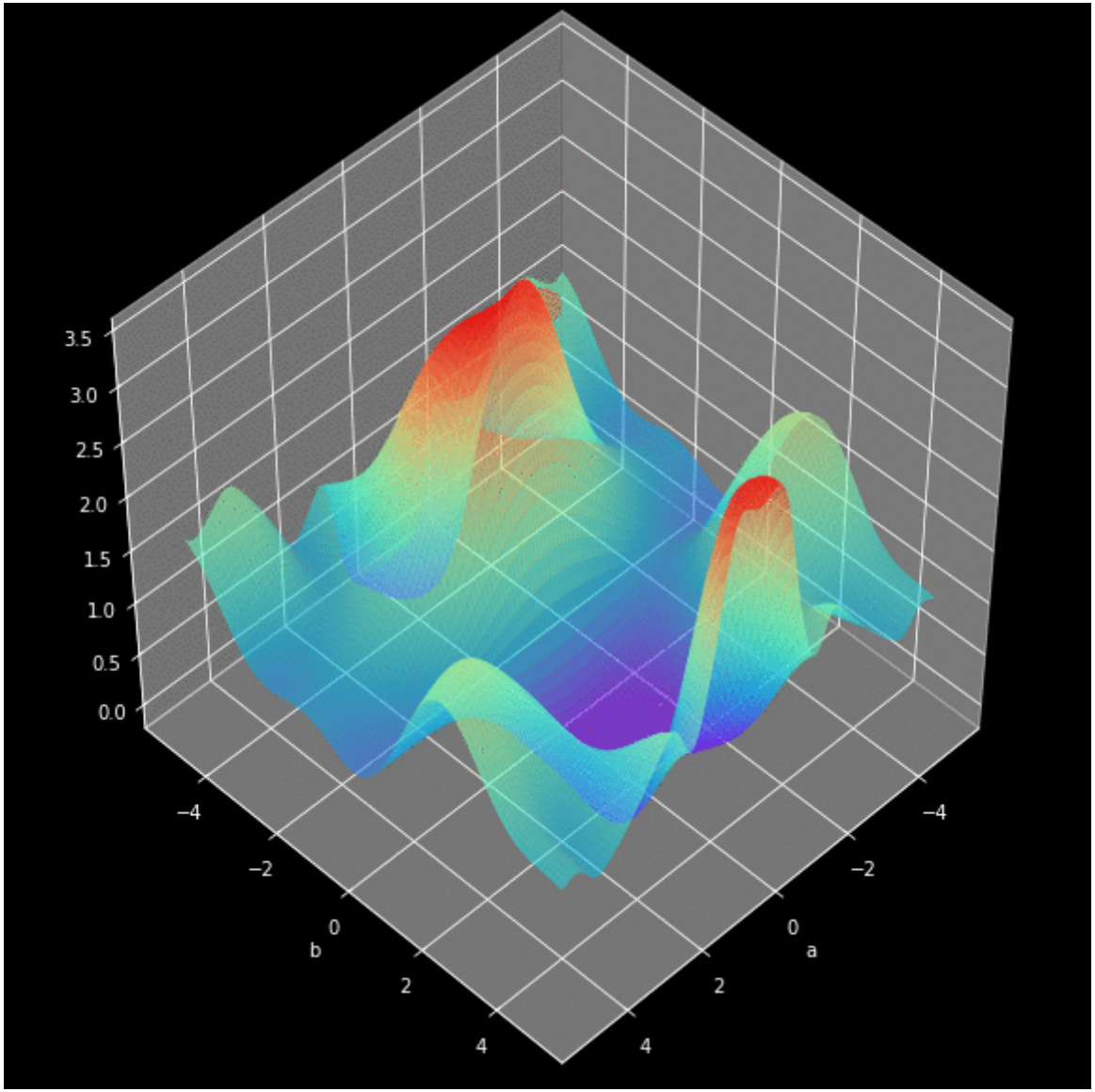


Figure 5.1 Visualization of gradient descent

Optimizers

There so many different Optimization Algorithms..

```
Gradient Descent Variants
    Batch Gradient Descent
    Stochastic Gradient Descent (SGD)
    Mini-Batch Gradient Descent
```

```
GD Optimizations
    Momentum
    Adagrad
    Adadelta
    RMSprop
    Adam
```

The main difference is in how much data we use to compute the gradient of the objective function

Gradient Descent Variants

Batch Gradient Descent

we need to calculate the gradients for the whole dataset to perform just one update:

$$\theta = \theta - \eta * \nabla_\theta J(\theta)$$

batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces

Stochastic Gradient Descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$

So:

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update

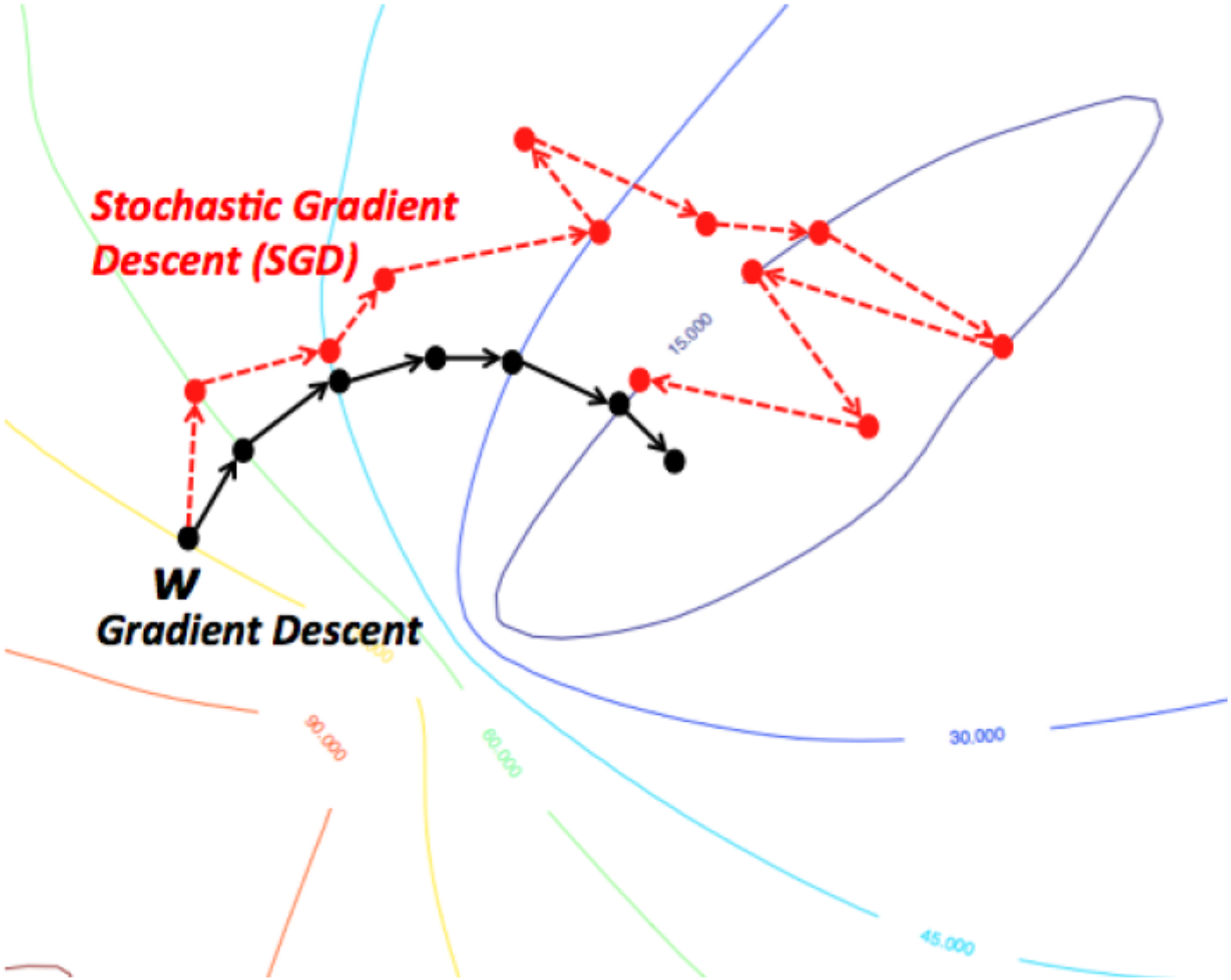


Figure 6.1 GD vs SGD

SGD performs frequent updates with a high variance that cause the objective function to fluctuate

Mini-Batch Gradient Descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

This produce:

```
Reduce Variance of parameter updates
More efficient Matrix multiplication
```

GD Optimizations

Momentum

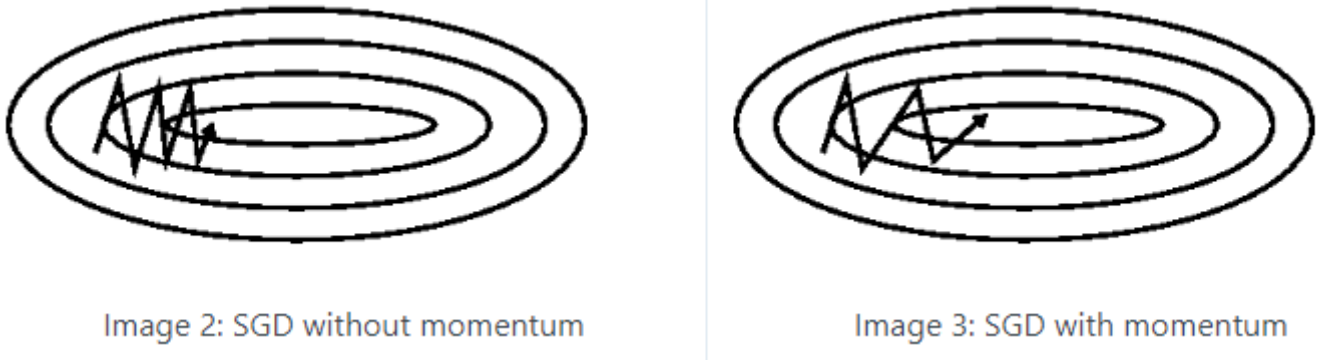


Figure 6.2 SGD | Momentum

Momentum helps the convergence by reducing the variance of parameters updates adding a fraction γ of the update vector of the past time step to the current update vector

So:

$$v_t = \gamma v_{t-1} + \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

$\gamma :\approx 0.9$

Adagrad

Adagrad s an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features:

$$Grad_{t,i} = \nabla_\theta J(\theta_{t,i})$$

So $\boldsymbol{\theta}_i$ becomes:

$$\theta_{t+1,i} = \theta t, i - \eta * Grad_{t,i}$$

In its update rule, Adagrad modifies the general learning rate ∇ at each time step t for every parameter $iin\theta$ based on the past gradients that have been computed for θ

$$\boldsymbol{\theta}_{t+1,i} = \boldsymbol{\theta}_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon} * Gradient_{t,i}$$

With $G_{t,ii}$ the diagonal matrix with the sum of squares of gradients in θ

$\eta : 0.01$

Adadelta & RMSprop

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w

Adam

Adaptive Moment Estimation similar to RMSprop & Adadelta but also keeps an exponentially decaying average of past gradients similar to momentum

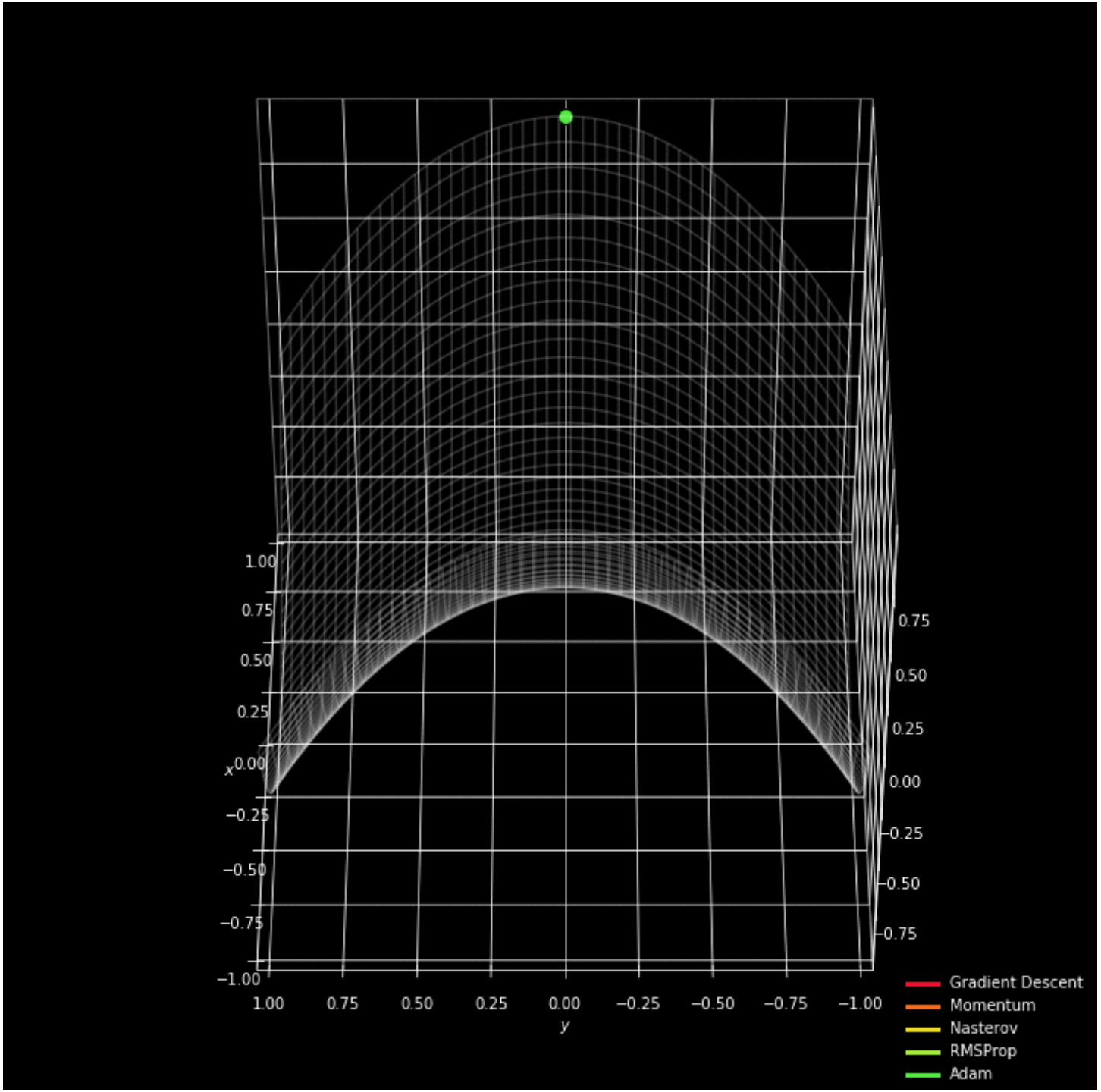

1.00
0.75
0.50
0.25
x 0.00
−0.25
−0.50
−0.75
−1.00
1.00 0.75 0.50 0.25 0.00 −0.25 −0.50 −0.75 −1.00
y
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Gradient Descent
Momentum
Nasterov
RMSProp
Adam

Figure 6.3 Optimizers Visualization

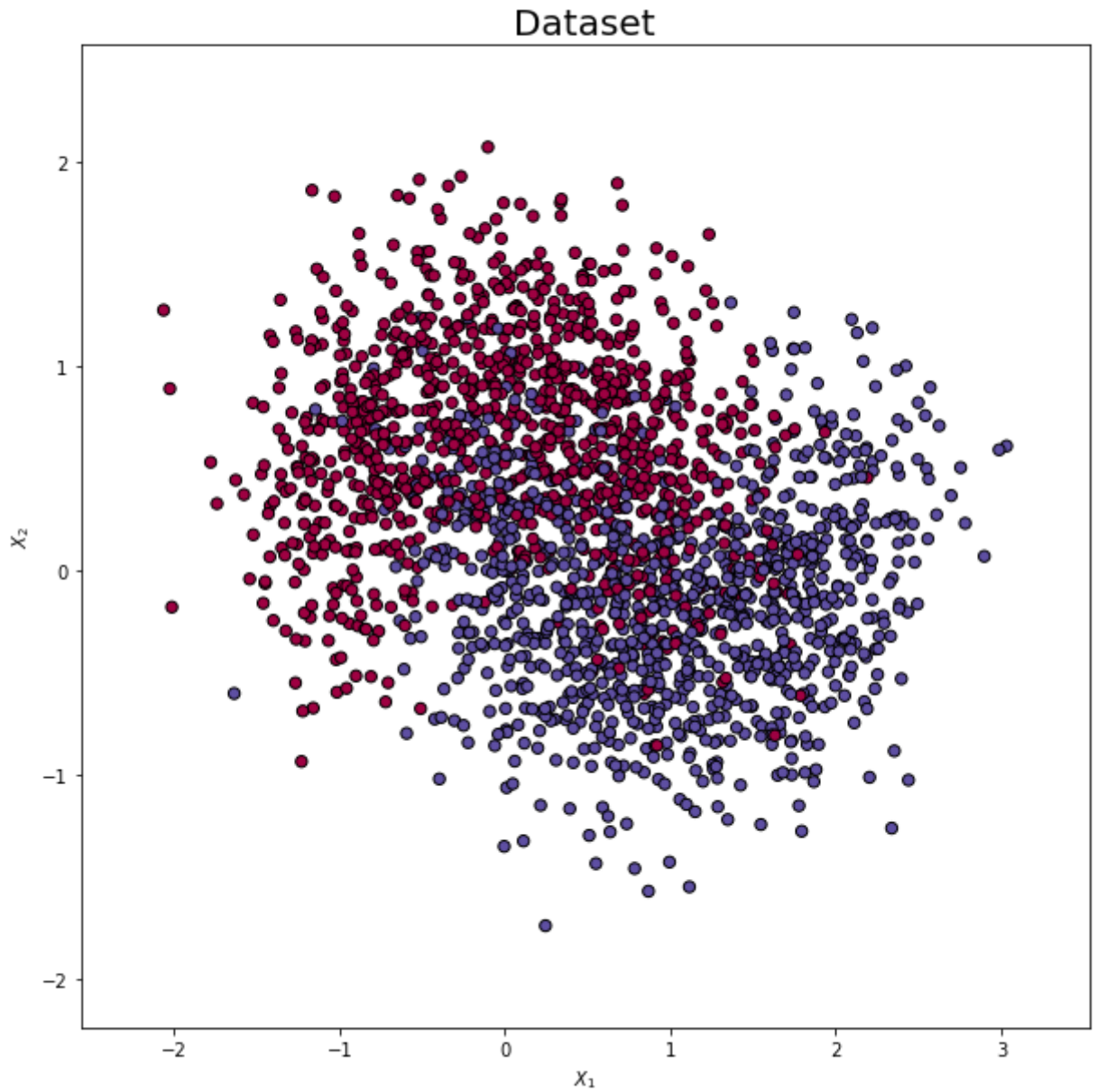
Summary

1) Structure of Perceptron
2) Forward & Backward Propagation
3) Learning Rate
4) Optimizers

Let's see a MLP architecture!

Generate Dataset!

Out[7]: <matplotlib.collections.PathCollection at 0x285330dcc50>

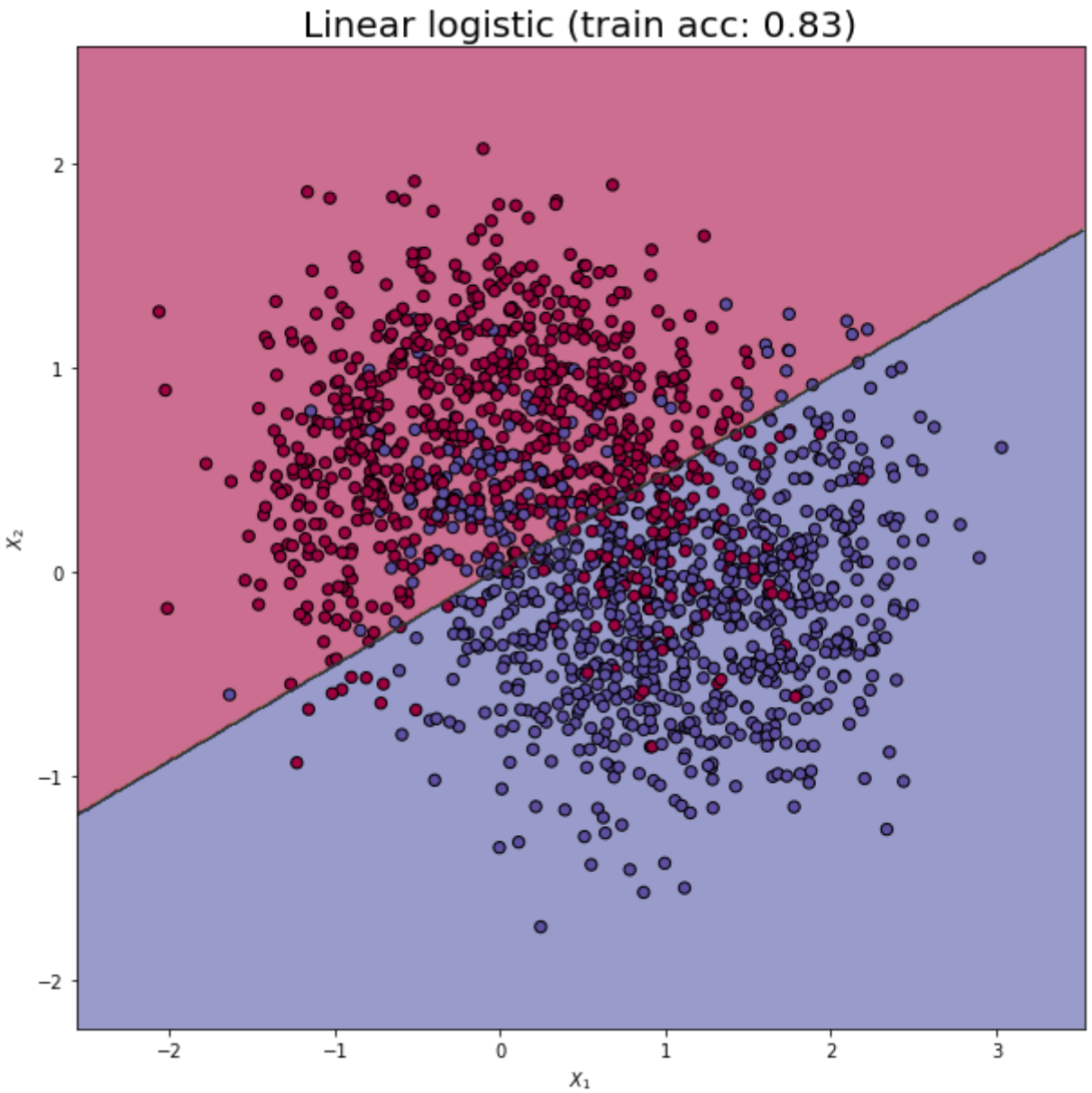


(1600, 2) (1600,) (400, 2) (400,)

Logistic regression

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 1)                 3         
=================================================================
Total params: 3.0
Trainable params: 3
Non-trainable params: 0.0
_________________________________________________________________
```

```
Train accuracy: 0.829375
Test accuracy: 0.85
```



Deeper Neural network

```
model_1 = Sequential()
model_1.add(Dense(8, input_dim=2,activation='relu'))
model_1.add(Dense(32, activation='relu'))
model_1.add(Dense(128, activation='relu'))
model_1.add(Dense(256, activation='relu'))
model_1.add(Dense(256, activation='relu'))
model_1.add(Dense(128, activation='relu'))
model_1.add(Dense(64, activation='relu'))
model_1.add(Dense(16, activation='relu'))
model_1.add(Dense(1, activation='sigmoid'))

model_1.compile(loss='binary_crossentropy', optimizer='adamax', metrics=['accuracy'])
model_1.summary()
print('loss: binary_crossentropy   ', 'optimizer: adamax    ', 'metrics: accuracy   ','activation: relu')
```

Figure 7 MLP

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_35 (Dense)             (None, 8)                 24        
_________________________________________________________________
dense_36 (Dense)             (None, 32)                288       
_________________________________________________________________
dense_37 (Dense)             (None, 128)               4224      
_________________________________________________________________
dense_38 (Dense)             (None, 256)               33024     
_________________________________________________________________
dense_39 (Dense)             (None, 256)               65792     
_________________________________________________________________
dense_40 (Dense)             (None, 128)               32896     
_________________________________________________________________
dense_41 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_42 (Dense)             (None, 16)                1040      
_________________________________________________________________
dense_43 (Dense)             (None, 1)                 17        
=================================================================
Total params: 145,561.0
Trainable params: 145,561
Non-trainable params: 0.0
_________________________________________________________________
loss: binary_crossentropy    optimizer: adamax     metrics: accuracy    activation: relu
```

Fit Network

```
Epoch 1/50
1600/1600 [==============================] - ETA: 0s - loss: 0.2880 - acc: 0.843 - ETA: 0s - loss: 0.3060 - acc: 0.
854 - ETA: 0s - loss: 0.3071 - acc: 0.856 - ETA: 0s - loss: 0.3068 - acc: 0.857 - ETA: 0s - loss: 0.3063 - acc: 0.8
65 - ETA: 0s - loss: 0.3035 - acc: 0.867 - ETA: 0s - loss: 0.3056 - acc: 0.864 - 0s - loss: 0.3013 - acc: 0.8656

Epoch 2/50
1600/1600 [==============================] - ETA: 0s - loss: 0.2093 - acc: 0.875 - ETA: 0s - loss: 0.2860 - acc: 0.
850 - ETA: 0s - loss: 0.2988 - acc: 0.856 - ETA: 0s - loss: 0.2930 - acc: 0.866 - ETA: 0s - loss: 0.3033 - acc: 0.8
69 - ETA: 0s - loss: 0.3020 - acc: 0.873 - ETA: 0s - loss: 0.3068 - acc: 0.870 - 0s - loss: 0.3059 - acc: 0.8669

Epoch 3/50
1600/1600 [==============================] - ETA: 0s - loss: 0.3308 - acc: 0.812 - ETA: 0s - loss: 0.2974 - acc: 0.
875 - ETA: 0s - loss: 0.2701 - acc: 0.877 - ETA: 0s - loss: 0.2862 - acc: 0.875 - ETA: 0s - loss: 0.2983 - acc: 0.8
65 - ETA: 0s - loss: 0.2976 - acc: 0.869 - ETA: 0s - loss: 0.3100 - acc: 0.860 - 0s - loss: 0.3087 - acc: 0.8600

Epoch 4/50
1600/1600 [==============================] - ETA: 0s - loss: 0.2135 - acc: 0.906 - ETA: 0s - loss: 0.2746 - acc: 0.
893 - ETA: 0s - loss: 0.2928 - acc: 0.884 - ETA: 0s - loss: 0.2836 - acc: 0.885 - ETA: 0s - loss: 0.2859 - acc: 0.8
84 - ETA: 0s - loss: 0.2948 - acc: 0.877 - ETA: 0s - loss: 0.3045 - acc: 0.869 - 0s - loss: 0.3049 - acc: 0.8675
```

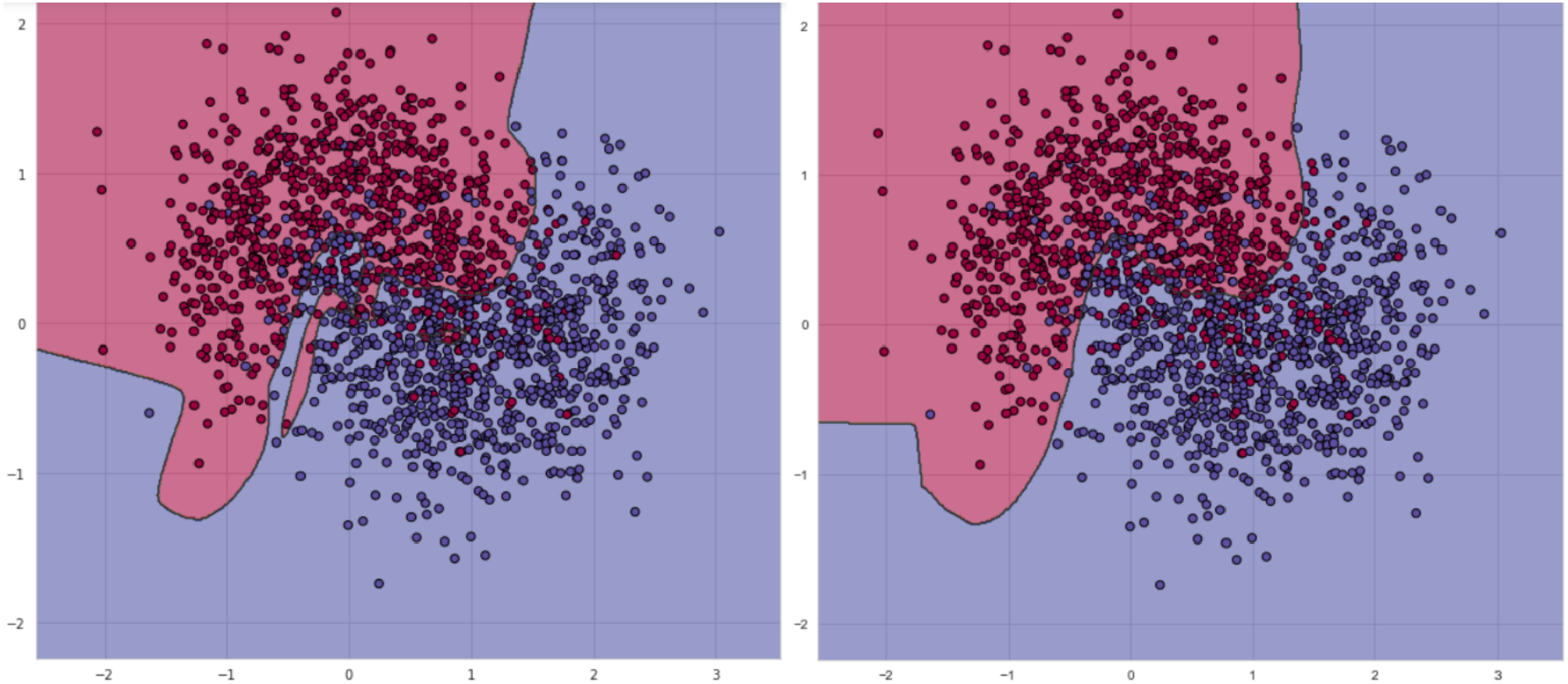


Figure 7.1 Train without | with Regularization

Regularization

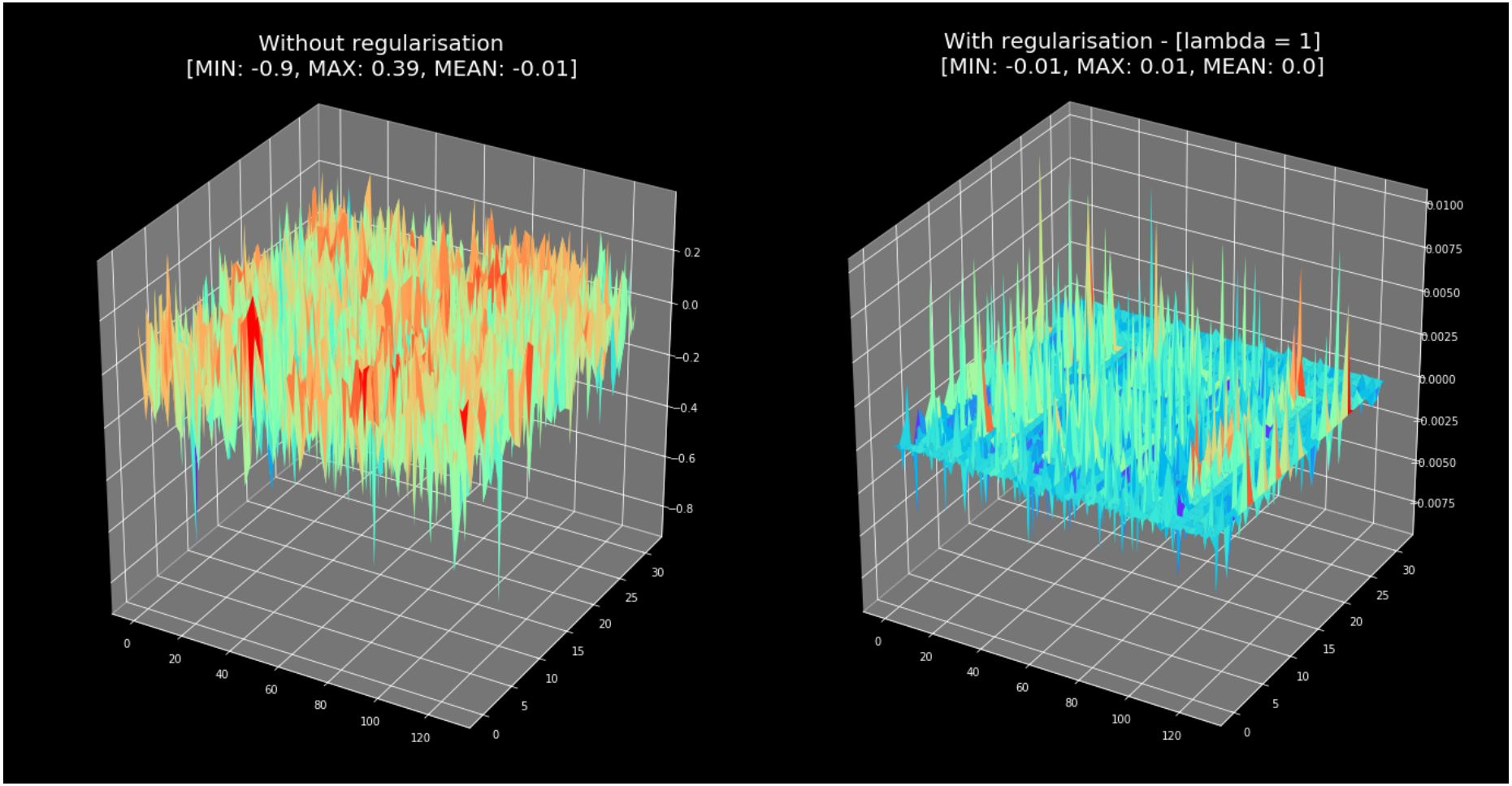


Figure 7 Visualize Regularization

Different Regularization techniques in Deep Learning

```
L2 and L1 regularization
Dropout
Data augmentation
Early stopping
```

L2 and L1 regularization

Values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent

$$CostFunction = Loss + \lambda$$

where λ is the penality term of Regularization

L2

$$CostFunction = Loss + \frac{\lambda}{2m} * \sum ||\boldsymbol{w}||^2$$

In L2 regularization we have a "weight decay" as it forces the weights to decay towards zero

L1

$$CostFunction = Loss + \frac{\lambda}{2m} * \sum ||w||$$

While here we penalize the absolute value of the weights, the weights may be reduced to zero.

- It is very useful when we are trying to compress our model

Dropout

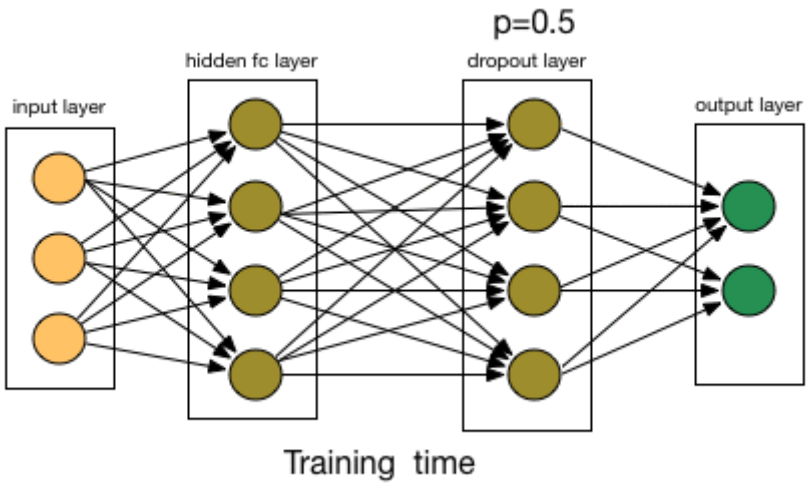


Figure 7.2 Dropout

Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness

Data augmentation

Figure 7.3 Data augmentation

Attention to Model Architecture!!

Early stopping

Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model

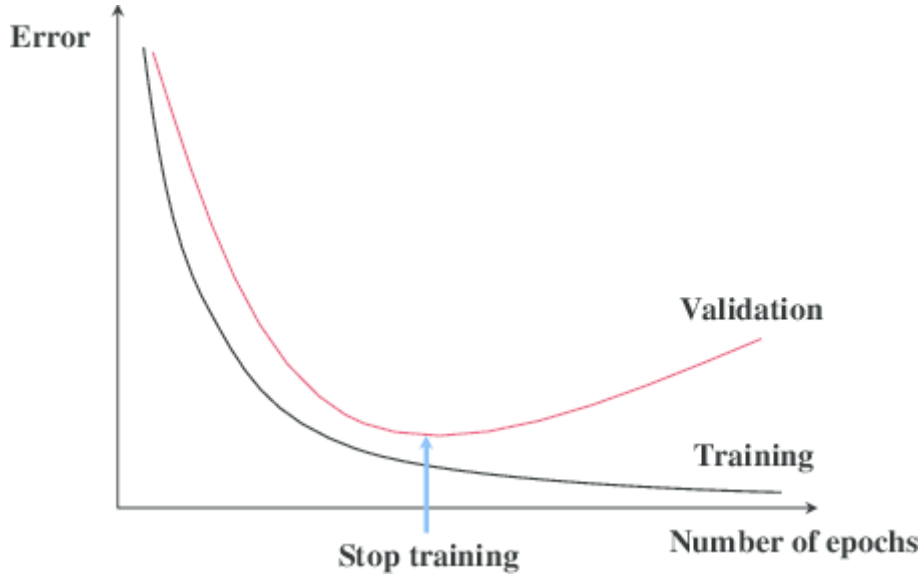
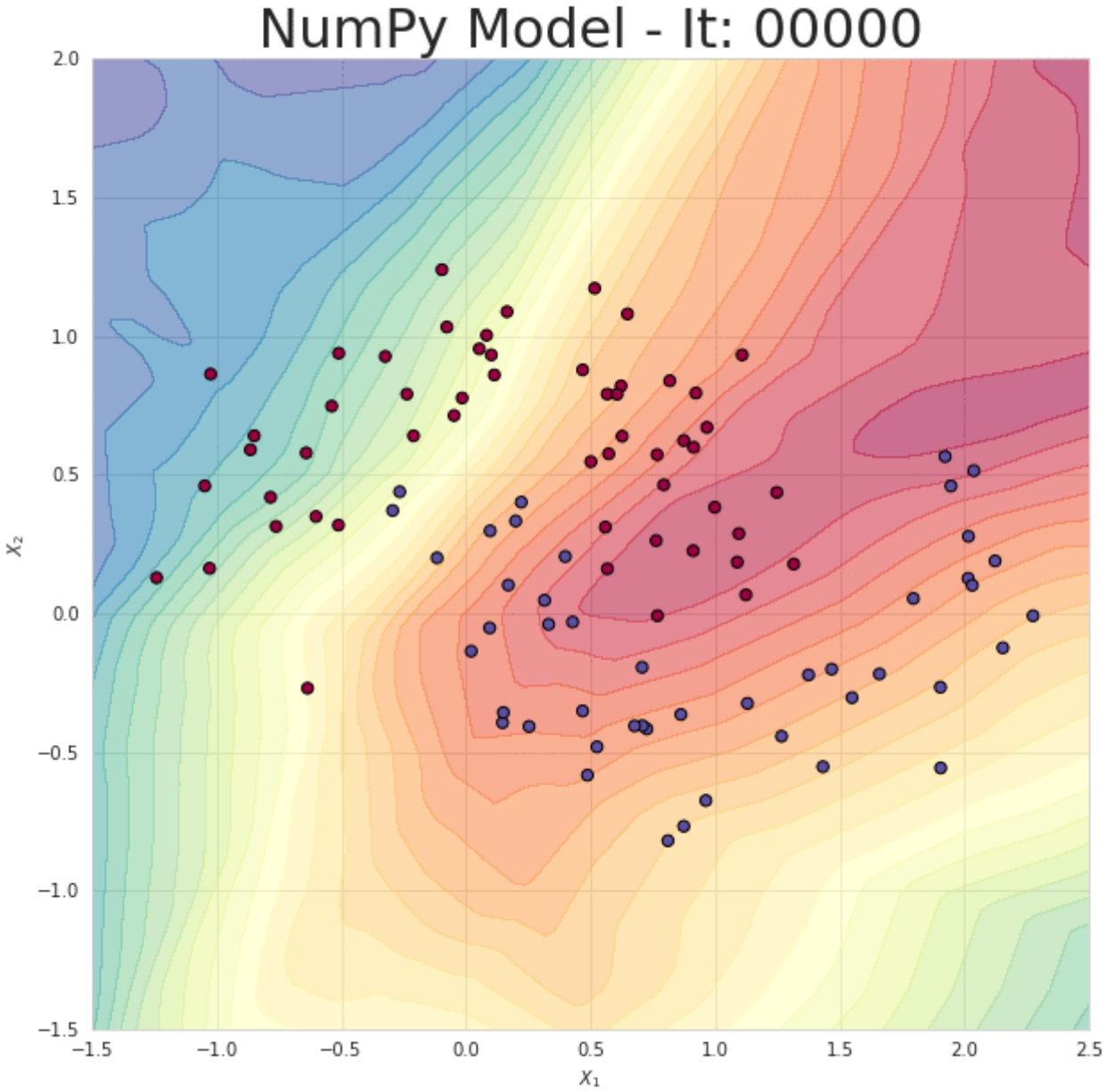


Figure 7.4 Early Stopping

Visualize Model Training


NumPy Model - It: 00000
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
X_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
X_1

Grid Search

```
def Talos_Grid_Search_DLI(x_train, y_train, x_val, y_val, params):

    model = Sequential()
    model.add(Dense(params['first_neuron'], input_dim=x_train.shape[1],
                    activation=params['activation'],
                    kernel_initializer=params['kernel_initializer']))

    model.add(Dropout(params['dropout']))

    model.add(Dense(1, activation=params['last_activation'],
                    kernel_initializer=params['kernel_initializer']))

    model.compile(loss=params['losses'],
                  optimizer=params['optimizer'](),
                  metrics=['acc', fmeasure_acc])

    history = model.fit(x_train, y_train,
                        validation_data=[x_val, y_val],
                        batch_size=params['batch_size'],
                        callbacks=[live()],
                        epochs=params['epochs'],
                        verbose=1)

    return history, model
```

Figure 8.1 Grid Search Architecture

```
p = {'first_neuron':[9,10,11],
     'hidden_layers':[0, 1, 2],
     'batch_size': [30],
     'epochs': [100],
     'dropout': [0],
     'kernel_initializer': ['uniform','normal'],
     'optimizer': [Nadam, Adam],
     'losses': [binary_crossentropy],
     'activation':[relu, elu],
     'last_activation': ['sigmoid']}
```