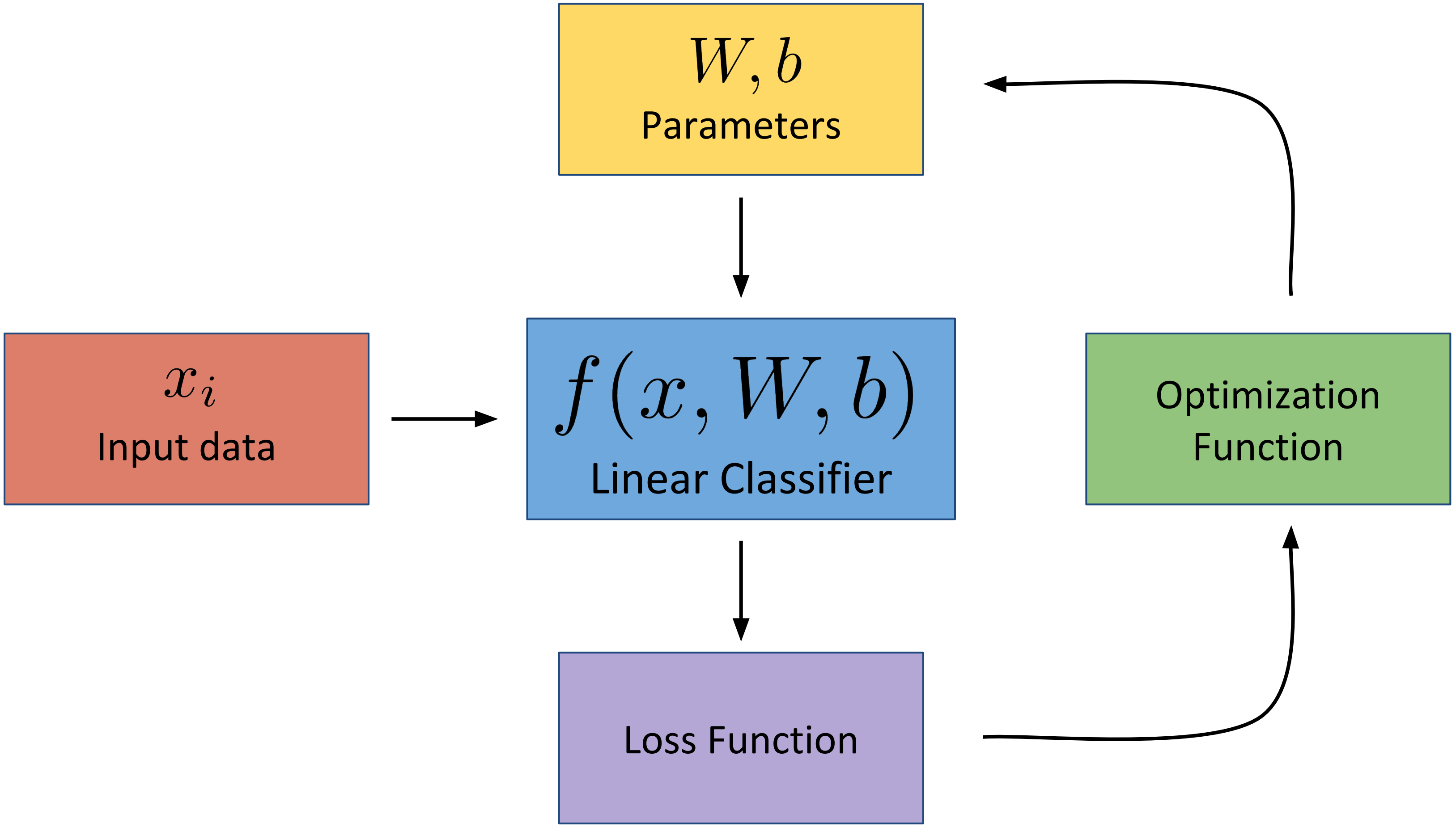
Course series: Deep Learning for Machine Translation

Neural Networks

Lecture # 5

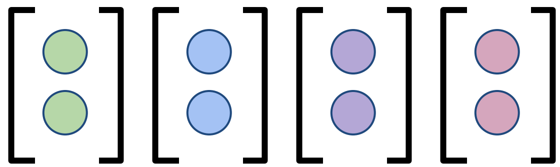
Hassan Sajjad and Fahim Dalvi
Qatar Computing Research Institute, HBKU

Recap



W, b
Parameters
x_i
Input data
$f(x, W, b)$
Linear Classifier
Optimization
Function
Loss Function

Data Representations

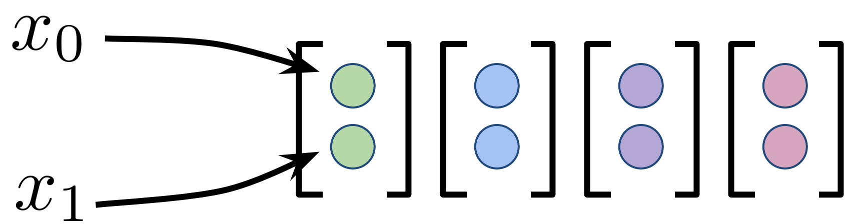


Dataset

4 examples

2 features

Data Representations



Dataset

4 examples

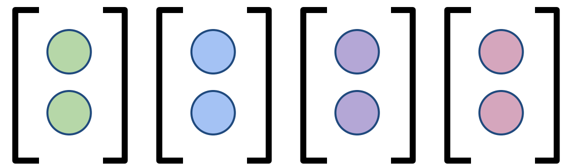
2 features

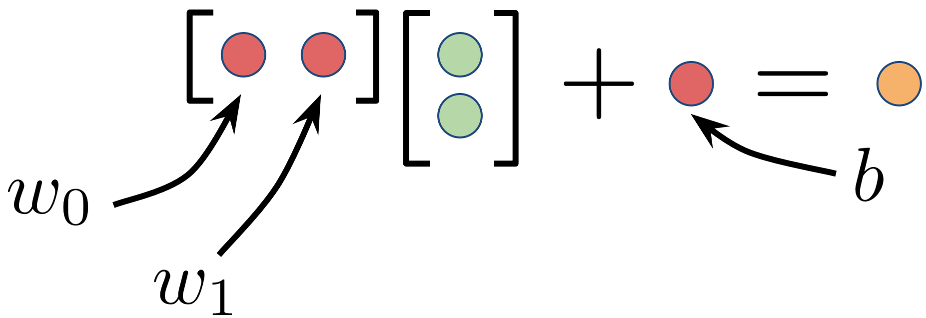
Data Representations

Dataset

4 examples

2 features





Vector

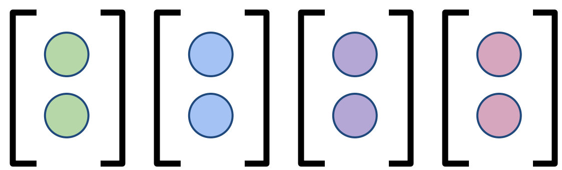
$$f(x, w, b) = \boxed{w} \cdot \boxed{x} + \boxed{b}$$

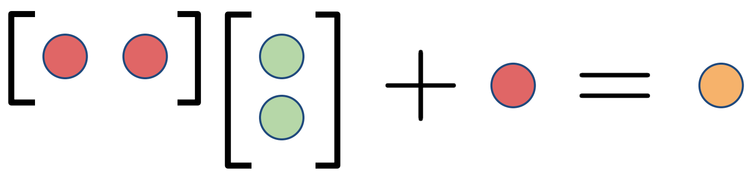
Real number

Linear Regression

Data Representations

Dataset
4 examples
2 features



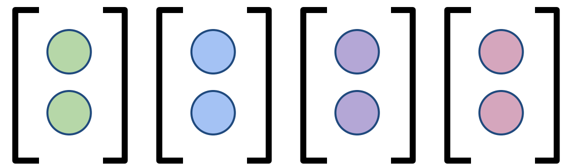


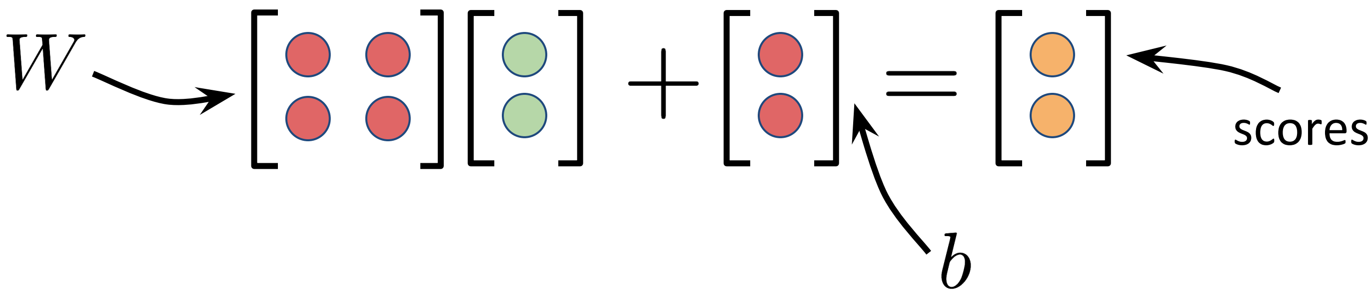
$$f(x, w, b) = w \cdot x + b$$

Linear Regression

Data Representations

Dataset
4 examples
2 features





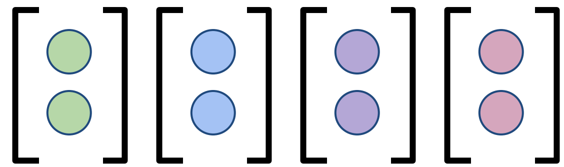
Matrix
Vector

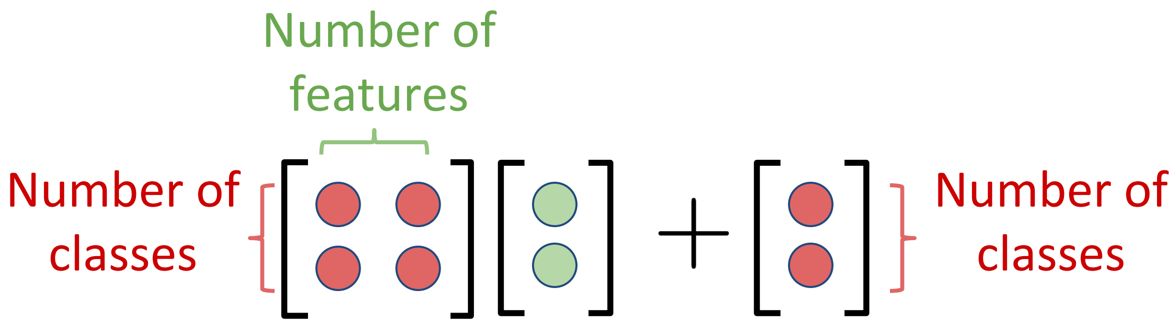
$$f(x, W, b) = \boxed{W} \cdot \boxed{x} + \boxed{b}$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features



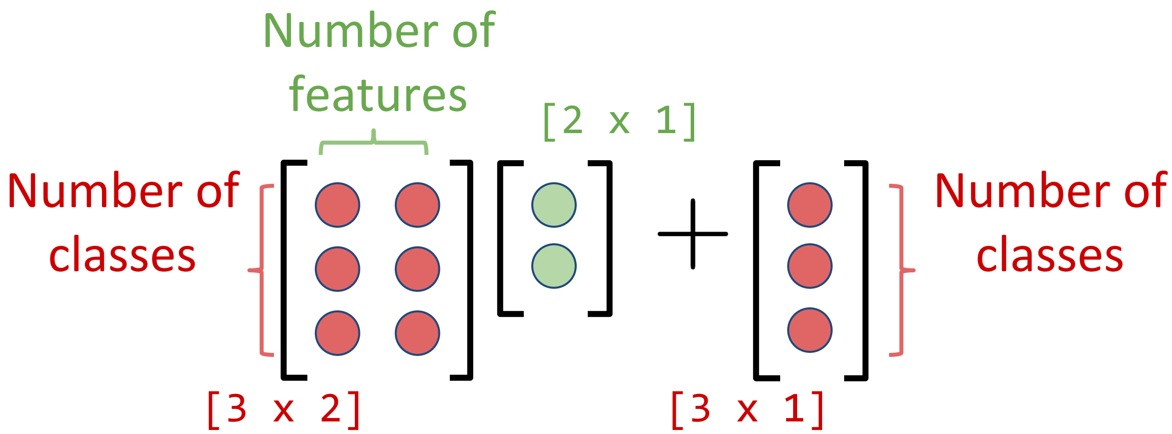


$$f(x, W, b) = \overset{\text{Matrix}}{\boxed{W}} \cdot \overset{\text{Vector}}{\boxed{x}} + \overset{\text{Vector}}{\boxed{b}}$$

Multi-class Linear Classification

Data Representations

3 class classification

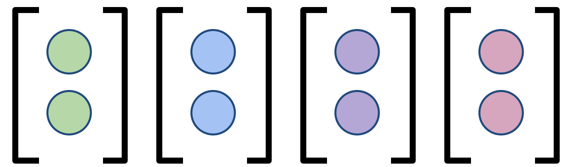


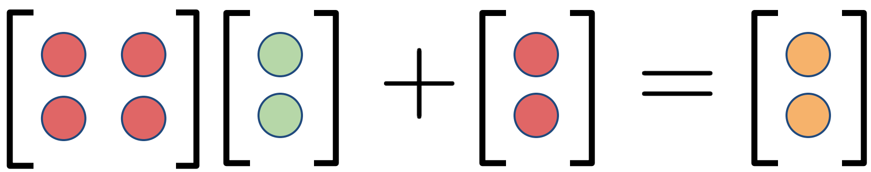
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features





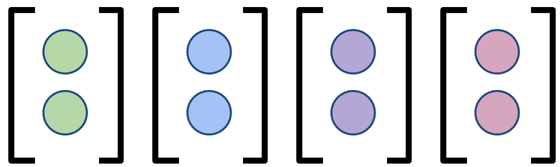
In this case, we are performing the above computation *per example*

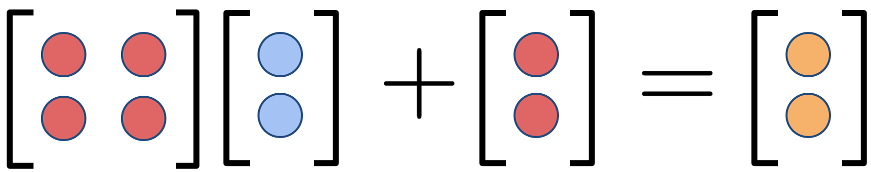
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features





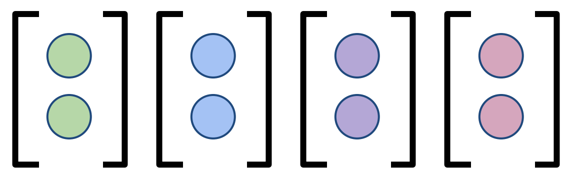
In this case, we are performing the above computation *per example*

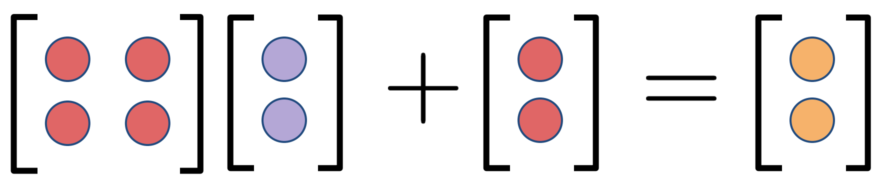
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features





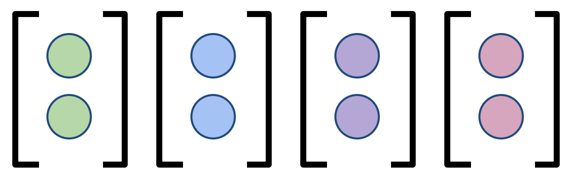
In this case, we are performing the above computation *per example*

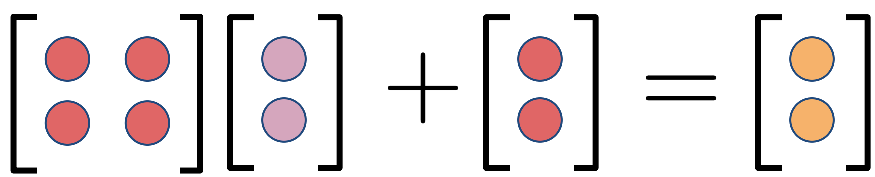
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features





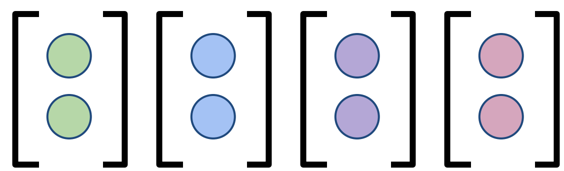
In this case, we are performing the above computation *per example*

$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features



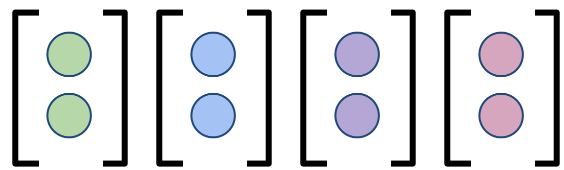
What if we can process all the examples in one go?

$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features



What if we can process all the examples in one go?

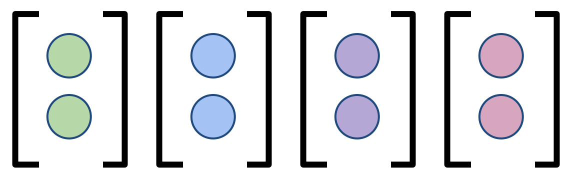
How: Stack all examples into one big matrix!

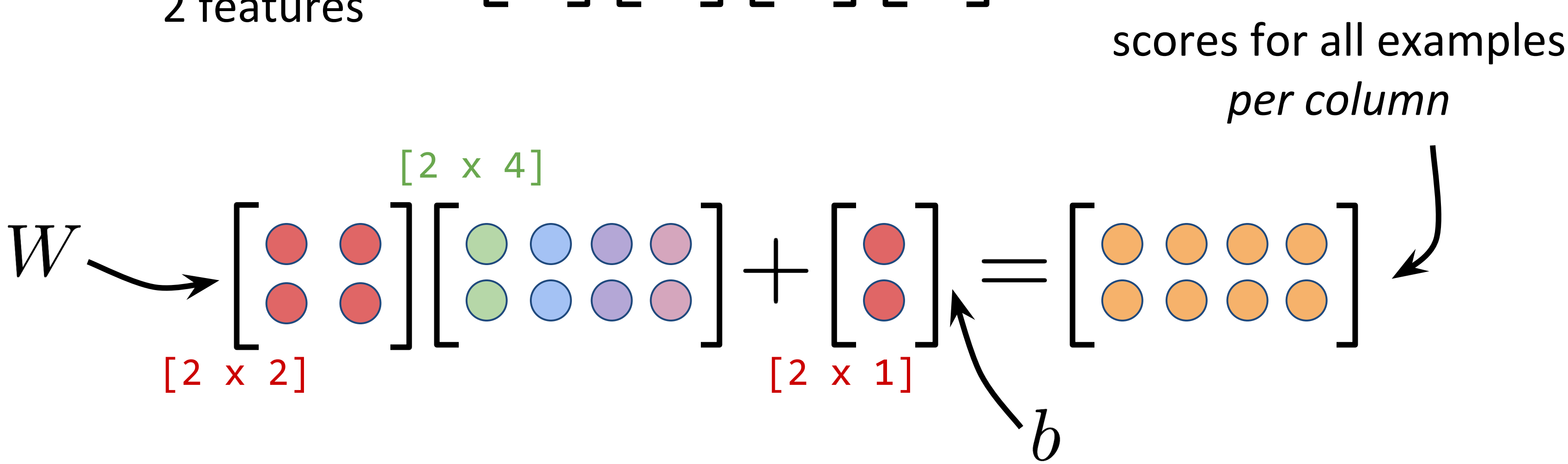
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

Data Representations

Dataset
4 examples
2 features





Matrix
Vector

$$f(X, W, b) = \boxed{W} \cdot \boxed{X} + \boxed{b}$$

Efficient Multi-class Linear Classification

Linear Classification with Softmax

Let us now bring it all together:

- Multiclass classification
- Batch Gradient Descent
- Softmax
- Cross Entropy Loss
- Regularization
- Efficient matrix multiplications

Linear Classification with Softmax

Reminder:

$$f(x, W, b) = W \cdot x + b$$

Objective

$$L = -\log(softmax(f)_c)$$

Loss

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

$$\frac{\partial L}{\partial b} = softmax(f) - \mathbb{I}(y = c)$$

Gradients

Softmax Linear Classifier

Total loss: average cross-entropy loss over the training examples and the regularization loss

$$L = \frac{1}{N}\sum_i L_i + \frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2$$

Reminder: Regularization

Because there are multiple possible solutions, we want to constrain the values of our parameters for better optimization

$$L = \texttt{Normal loss} + \lambda \sum_{w} w_i^2$$

Penalizes weights that are too large
λ defines how much importance you want to give to regularization

Softmax Linear Classifier

```
# initialize parameters randomly
W = 0.01 * np.random.random((num_features,num_classes))
b = np.random.random((1,num_classes))
```

```
# initialize hyperparameters
lr = 1e-0
reg = 1e-3 # regularization strength
```

Parameter Initialization
Remember that w is a matrix now, and b is a vector. We initialize parameters randomly

Softmax Linear Classifier

```
# initialize parameters randomly
W = 0.01 * np.random.random((num_features,num_classes))
b = np.random.random((1,num_classes))
```

```
# initialize hyperparameters
lr = 1e-0
reg = 1e-3 # regularization strength
```

Parameter Initialization
Remember that `W` is a matrix now, and b is a vector. We initialize parameters randomly

Note that we multiply `W` with `0.01` to make its values small - initializing with small random values works better in practice

Softmax Linear Classifier

```
# initialize parameters randomly
W = 0.01 * np.random.random((num_features,num_classes))
b = np.random.random((1,num_classes))
```

```
# initialize hyperparameters
lr = 1e-0
reg = 1e-3 # regularization strength
```

Hyperparameter Initialization: Note that in addition to the learning rate for gradient descent, we also set the regularization parameter for our updated loss function

Softmax Linear Classifier

```python
def fn(X, W, b):
    return np.dot(X, W) + b
```

```python
def softmax(f):
    exp_f = np.exp(f)
    probs = exp_f / np.sum(exp_f, axis=1, keepdims=True) # [num_examples x num_classes]

    return probs
```

```python
def loss(X, W, b, y):
    # compute the class probabilities
    probs = softmax(fn(X, W, b))

    # compute the loss: average cross-entropy loss and regularization
    correct_logprobs = -np.log(probs[range(num_examples),y])

    data_loss = np.sum(correct_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W)
    loss = data_loss + reg_loss
    return loss
```

Softmax Linear Classifier

```python
def fn(X, W, b):
    return np.dot(X, W) + b
```

Objective function

```python
def softmax(f):
    exp_f = np.exp(f)
    probs = exp_f / np.sum(exp_f, axis=1, keepdims=True) # [num_examples x num_classes]

    return probs
```

$$f(X, W, b) = X \cdot W + b$$

```python
def loss(X, W, b,
    # compute the
    probs = softm

    # compute the loss: average cross-entropy loss and regularization
    correct_logprobs = -np.log(probs[range(num_examples),y])

    data_loss = np.sum(correct_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W)
```

Here we use numpy's matrix operations to compute scores for all of the examples!

Output shape: [`num_examples x num_classes`]

Softmax Linear Classifier

```
def fn(X, W, b):
    return np.dot(X, W) + b
```

Softmax function

```
def softmax(f):
    exp_f = np.exp(f)
    probs = exp_f / np.sum(exp_f, axis=1, keepdims=True) # [num_examples x num_classes]

    return probs
```

```
def loss(X, W, b, y):
    # compute the
    probs = softma

    # compute the
    correct_logpro

    data_loss = np
    reg_loss = 0.5
    loss = data_lo
    return loss
```

$$p_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

Again, we use matrix operations to compute softmax for all examples simultaneously!

Softmax Linear Classifier

```
def fn(X, W, b):
    return np.dot(X, W) + b
```

```
def softmax(f):
    exp_f = np.
    probs = exp_                                                                 lasses]

    return prob
```

$$L_D = -\log(softmax(f)_c)$$

Loss computation

```
def loss(X, W, b, y):
    # compute the class probabilities
    probs = softmax(fn(X, W, b))

    # compute the loss: average cross-entropy loss and regularization
    correct_logprobs = -np.log(probs[range(num_examples),y])

    data_loss = np.sum(correct_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W)
    loss = data_loss + reg_loss
    return loss
```

Loss of every example is computed as the log probability of the correct class for that example

Softmax Linear Classifier

```python
def fn(X, W, b):
    return np.dot(X, W) + b
```

```python
def softmax
    exp_f =
    probs =

    return
```

$$L = \frac{1}{N} L_D + \frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2$$

```python
def loss(X,
    # compu
    probs =

    # compute the loss: average cross-entropy loss and regularization
    correct_logprobs = -np.log(probs[range(num_examples),y])

    data_loss = np.sum(correct_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W)
    loss = data_loss + reg_loss
    return loss
```

Total loss computation

Loss of every example is computed as the log probability of the correct class for that example

Softmax Linear Classifier

Implementing gradient functions

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

$$\frac{\partial L}{\partial b} = softmax(f) - \mathbb{I}(y = c)$$

The gradient for this classifier is a bit involved, but try and derive it analytically/by backpropagation yourself!

Softmax Linear Classifier

Implementing gradient functions

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

$$\frac{\partial L}{\partial b} = softmax(f) - \boxed{\mathbb{I}(y = c)}$$

Indicator function - its value is 1 if the condition within the parenthesis is true, 0 otherwise

Softmax Linear Classifier

Implementing gradient functions

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

$$\frac{\partial L}{\partial b} = softmax(f) - \mathbb{I}(y = c)$$

Intuition: If your softmax scores were `[0.2,0.3,0.5]` with the middle class as the correct class, the gradients would change the scores to `[0.2,-0.7,0.5]`

Softmax Linear Classifier

Implementing gradient functions

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

$$\frac{\partial L}{\partial b} = softmax(f) - \mathbb{I}(y = c)$$

```
       f = [ 0.2 ,  0.3 , 0.5 ]
gradient = [ 0.2 , -0.7 , 0.5 ]
```

Increasing corresponding scores in f will result in ***an increase*** in overall loss

Increasing corresponding score in f will result in ***a decrease*** in overall loss

Softmax Linear Classifier

```
def dfn_W(X, W, b, y):
    # compute the class probabilities
    probs = softmax(fn(X, W, b))

    # compute the gradient on scores
    df = probs
    df[range(num_examples),y] -= 1
    df /= num_examples

    # backpropate the gradient to the parameter W
    dW = np.dot(X.T, df)
    dW += reg*W # regularization gradient

    return dW
```

Backpropagation: here we subtract 1 from the probabilities of the correct class

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

Softmax Linear Classifier

```
def dfn_W(X, W, b, y):
    # compute the class probabilities
    probs = softmax(fn(X, W, b))

    # compute the gradient on scores
    df = probs
    df[range(num_examples),y] -= 1
    df /= num_examples

    # backpropate the gradient to the parameter W
    dW = np.dot(X.T, df)
    dW += reg*W # regularization gradient

    return dW
```

Backpropagation for weights W

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

Softmax Linear Classifier

```
def dfn_W(X, W, b, y):
    # compute the class probabilities
    probs = softmax(fn(X, W, b))

    # compute the gradient on scores
    df = probs
    df[range(num_examples),y] -= 1
    df /= num_examples

    # backpropate the gradient to the parameter W
    dW = np.dot(X.T, df)
    dW += reg*W # regularization gradient

    return dW
```

Add regularization gradient for W

$$\frac{\partial L}{\partial W} = \Big(softmax(f) - \mathbb{I}(y = c)\Big) \cdot x$$

Softmax Linear Classifier

```
# gradient descent loop
num_examples = X.shape[0]
for i in xrange(100):
    average_loss = loss(X, W, b, y)

    if i % 10 == 0:
        print "Epoch %d, loss %f" % (i, average_loss)

    dW = dfn_W(X, W, b, y)
    db = dfn_b(X, W, b, y)

    ### perform a parameter update
    W += -lr * dW
    b = b - lr * db
```

Main optimization loop
#epochs = 100

Softmax Linear Classifier

```
# gradient descent loop
num_examples = X.shape[0]
for i in xrange(100):
    average_loss = loss(X, W, b, y)

    if i % 10 == 0:
        print "Epoch %d, loss %f" % (i, average_loss)

    dW = dfn_W(X, W, b, y)
    db = dfn_b(X, W, b, y)

    ### perform a parameter update
    W += -lr * dW
    b = b - lr * db
```

Print average loss

This should go down as we train

Softmax Linear Classifier

```
# gradient descent loop
num_examples = X.shape[0]
for i in xrange(100):
    average_loss = loss(X, W, b, y)

    if i % 10 == 0:
        print "Epoch %d, loss %f" % (i, average_loss)

    dW = dfn_W(X, W, b, y)
    db = dfn_b(X, W, b, y)

    ### perform a parameter update
    W += -lr * dW
    b = b - lr * db
```

Compute gradients for the parameters over the entire set

Softmax Linear Classifier

```
# gradient descent loop
num_examples = X.shape[0]
for i in xrange(100):
    average_loss = loss(X, W, b, y)

    if i % 10 == 0:
        print "Epoch %d, loss %f" % (i, average_loss)

    dW = dfn_W(X, W, b, y)
    db = dfn_b(X, W, b, y)

    ### perform a parameter update
    W += -lr * dW
    b = b - lr * db
```

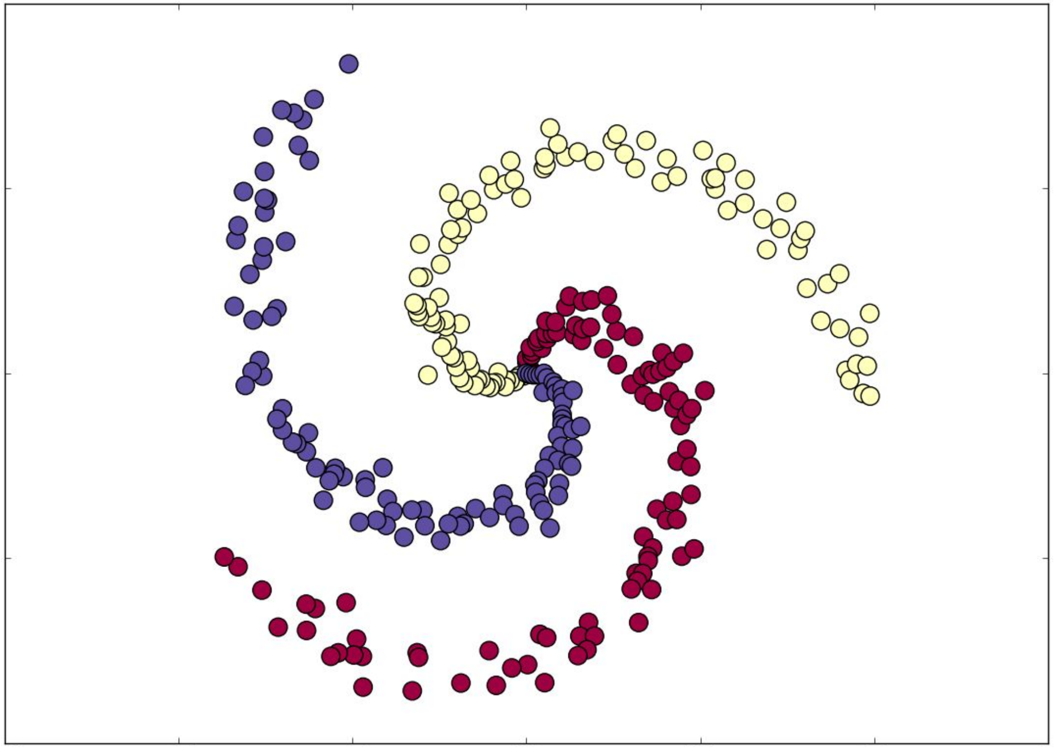
Update the parameters

Softmax Linear Classifier

Let’s run it!

Softmax Linear Classifier

Let’s now try the classifier on data that is *not linearly separable.*



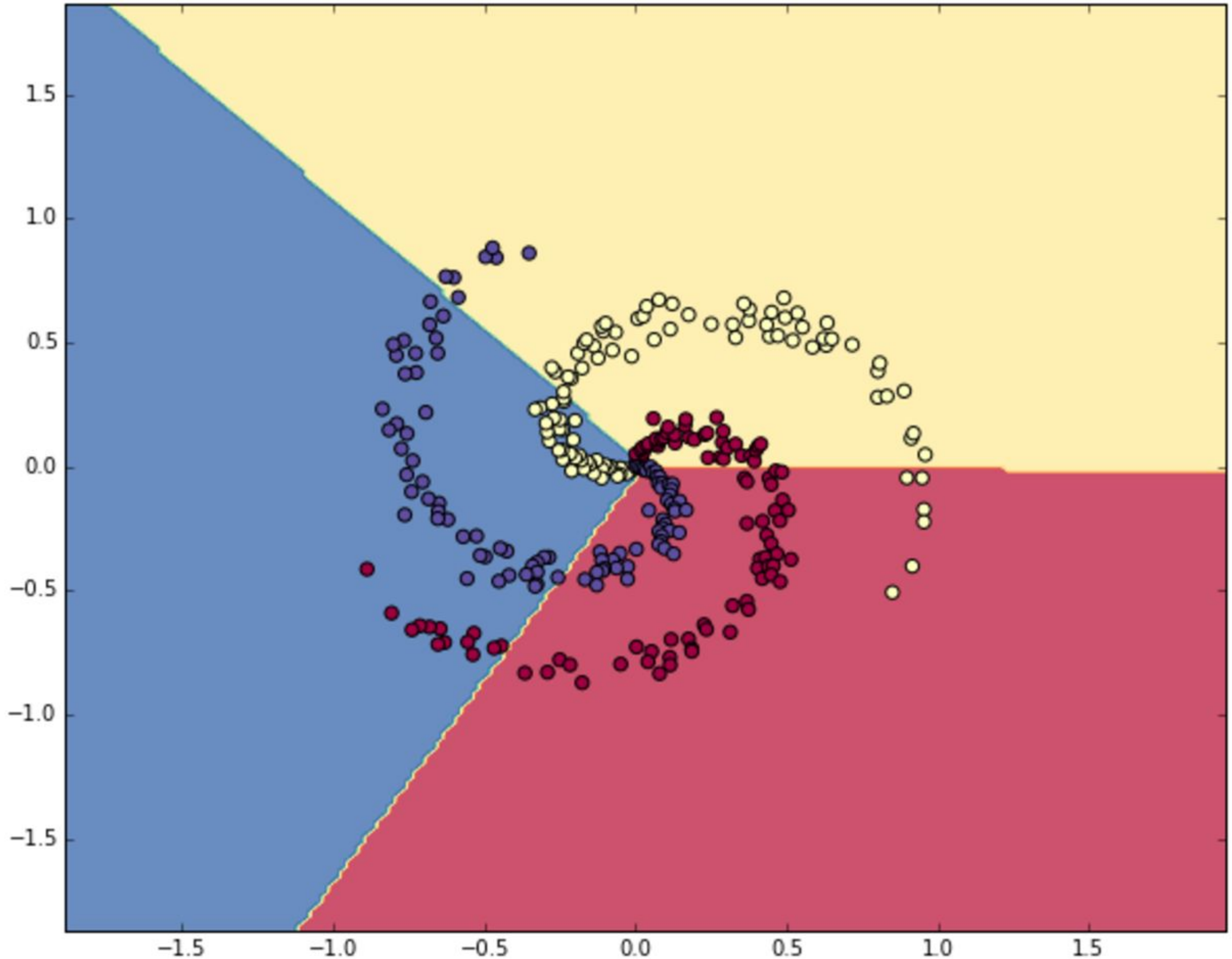
Softmax Linear Classifier

We achieve an accuracy of around 50% on the spiral data - Why?

Softmax Linear Classifier

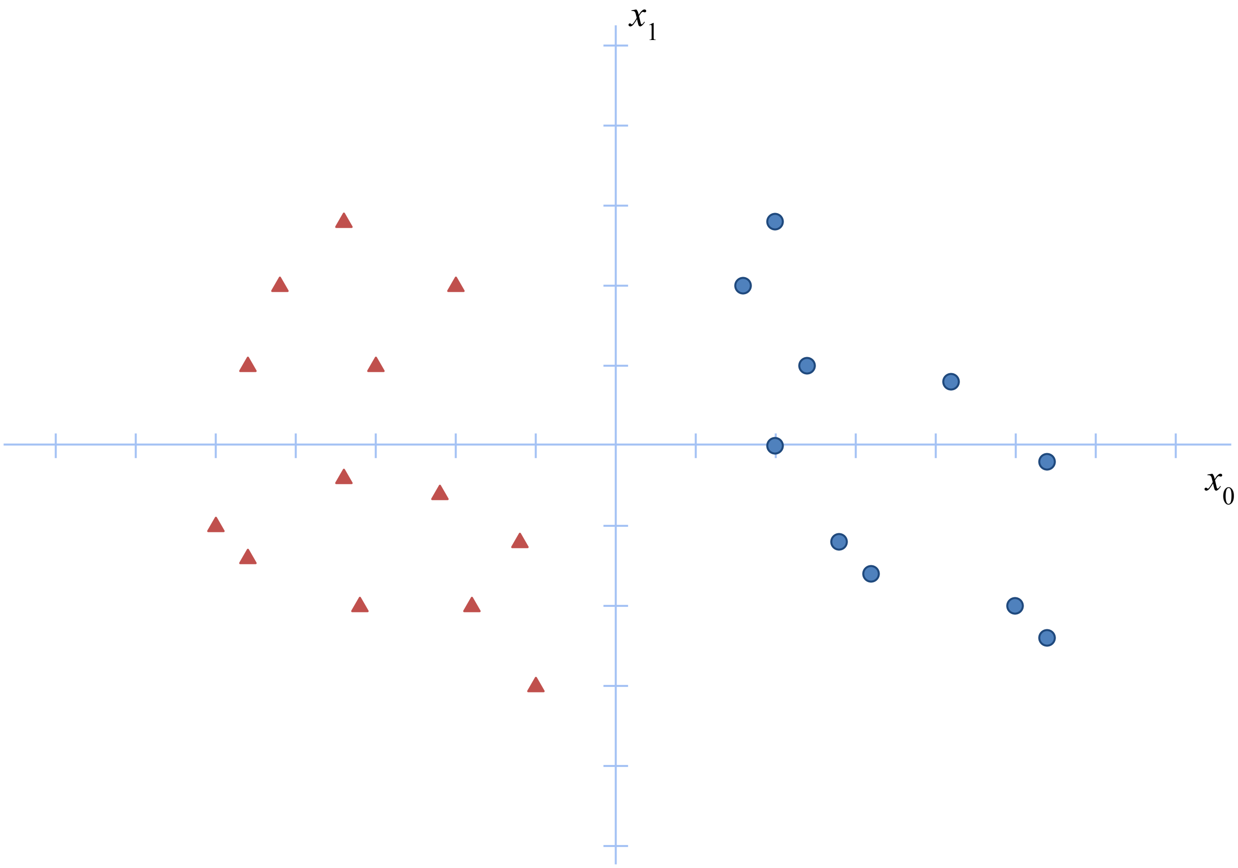
Not Linearly separable

Current models can only draw linear boundaries!

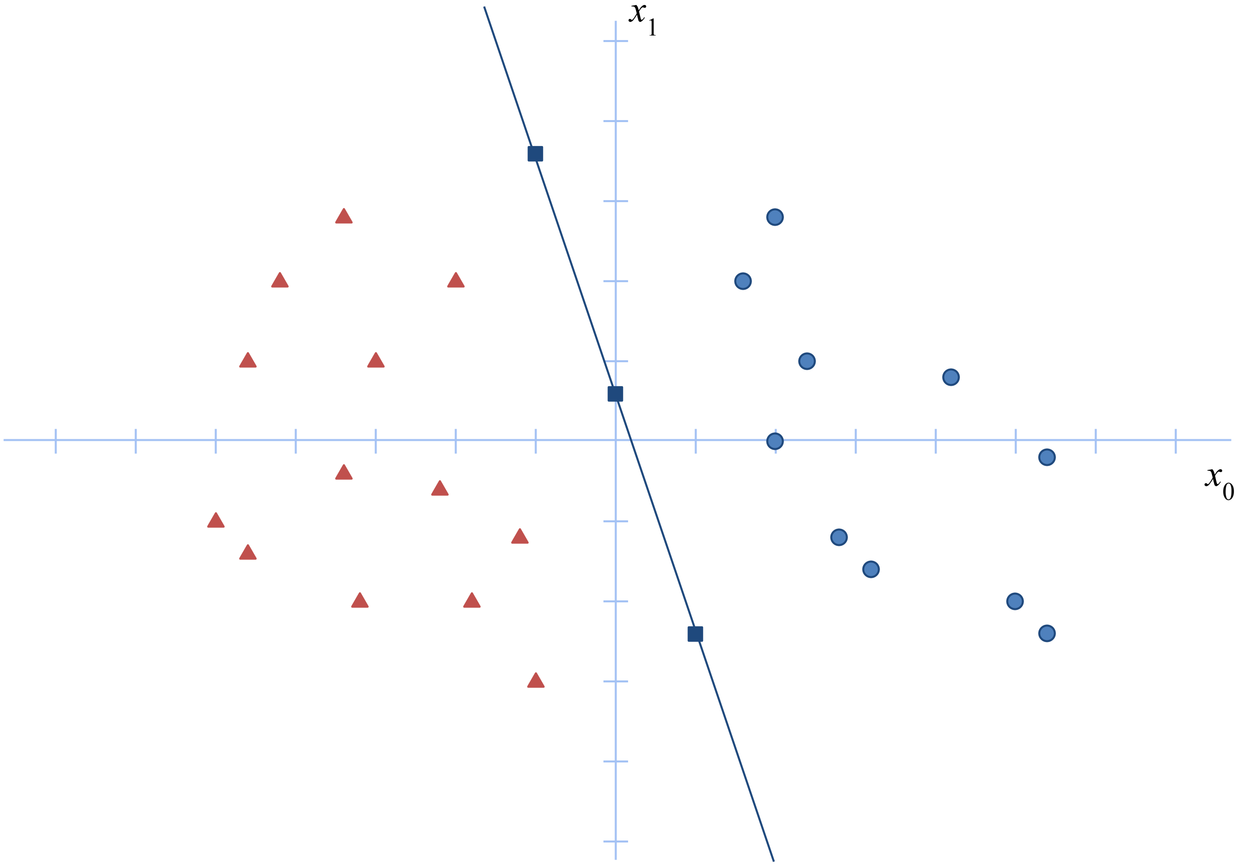


Neural Networks!

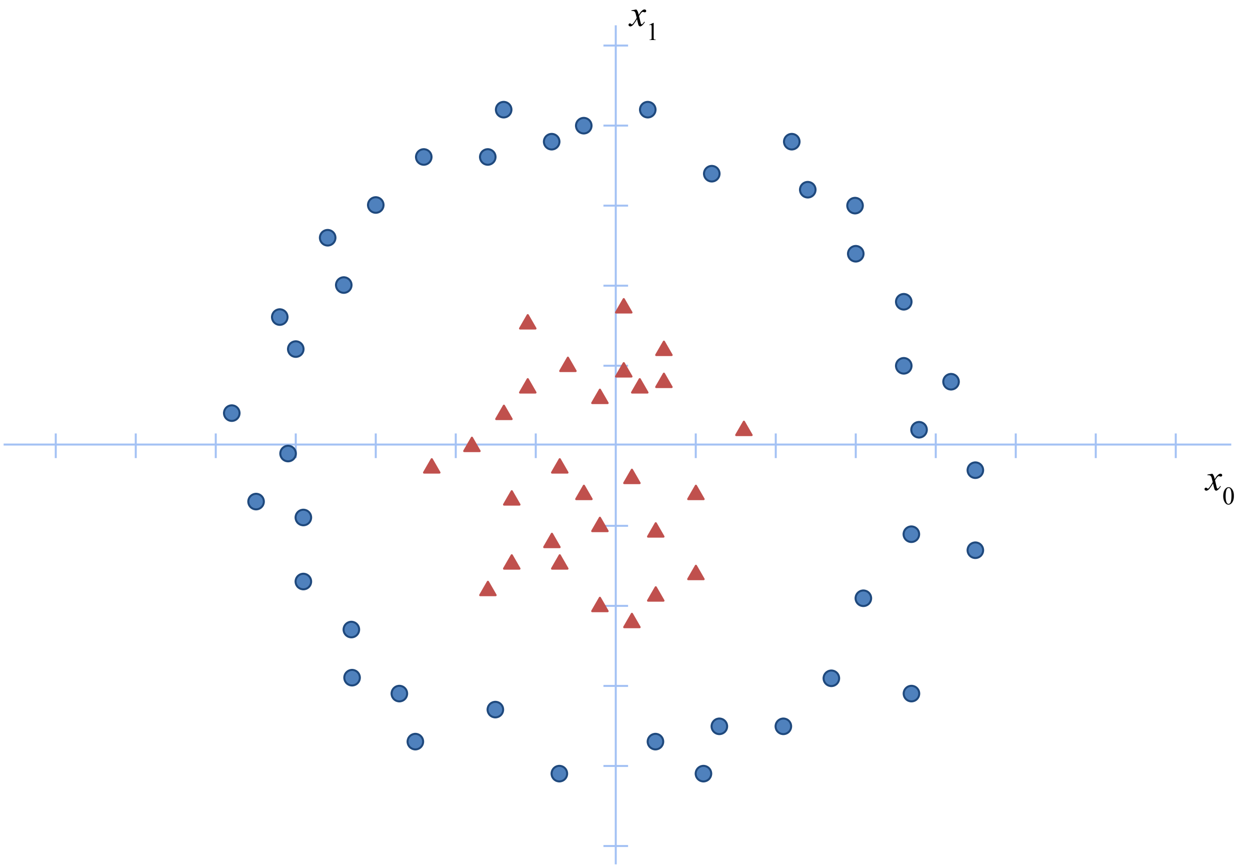
Linear Classifier Recap



Linear Classifier Recap



Linear Classifier?



Linear Separability

Not all problems are linearly classifiable - i.e. if you plot the examples in space, you cannot draw a line/plane to separate them out

Linear Separability

Not all problems are linearly classifiable - i.e. if you plot the examples in space, you cannot draw a line/plane to separate them out

Neural Networks are one way to solve this problem

Linear Classifier

$$f(x, W, b)$$

Linear Classifier

$$[0.3 \quad 1.2]$$

Score for the car to be accident prone

Score for the car to be *NOT* accident prone

Neural Network

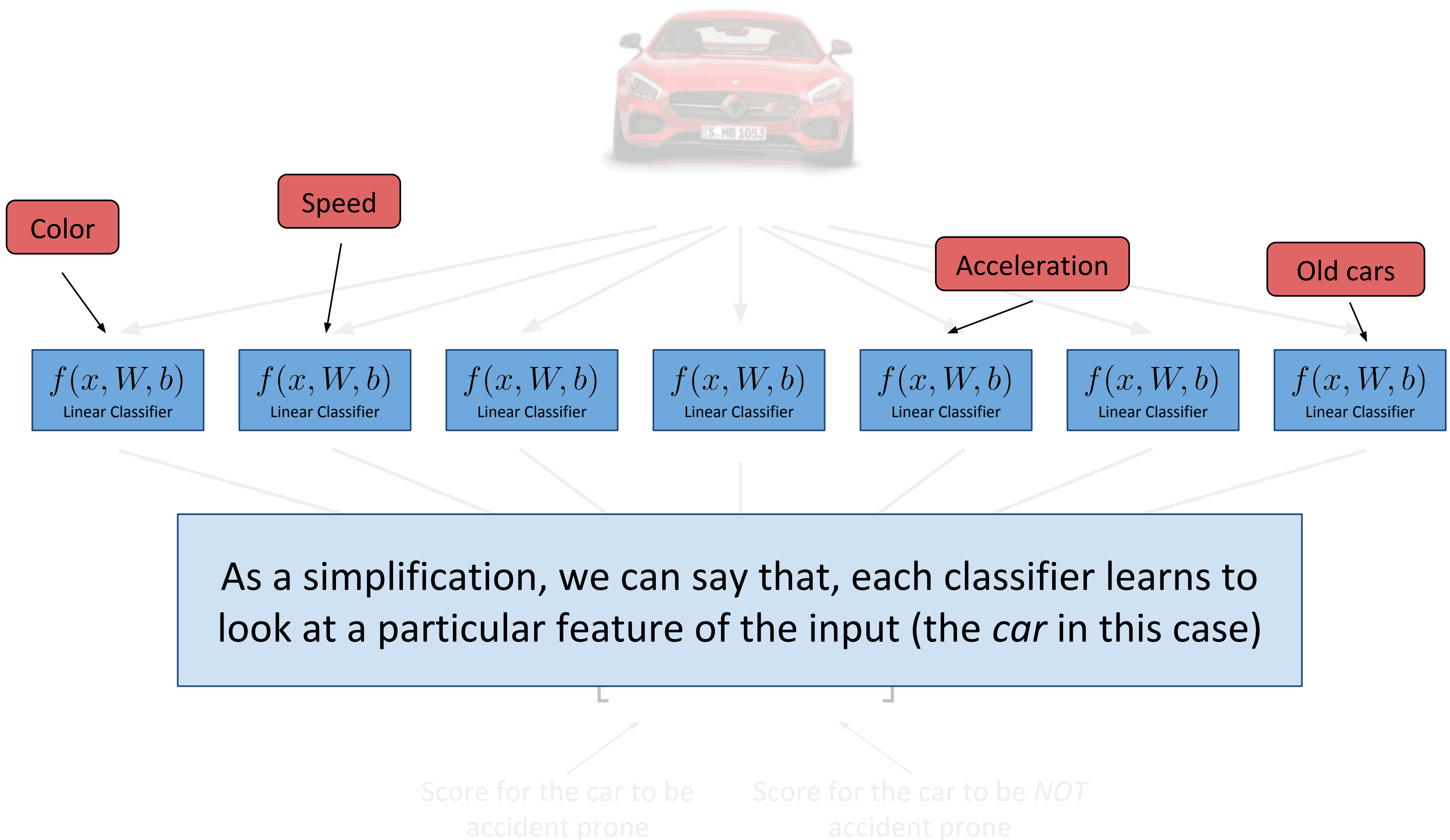
$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier	$f(x, W, b)$ Linear Classifier

$$[0.3 \quad 1.2]$$

Score for the car to be accident prone

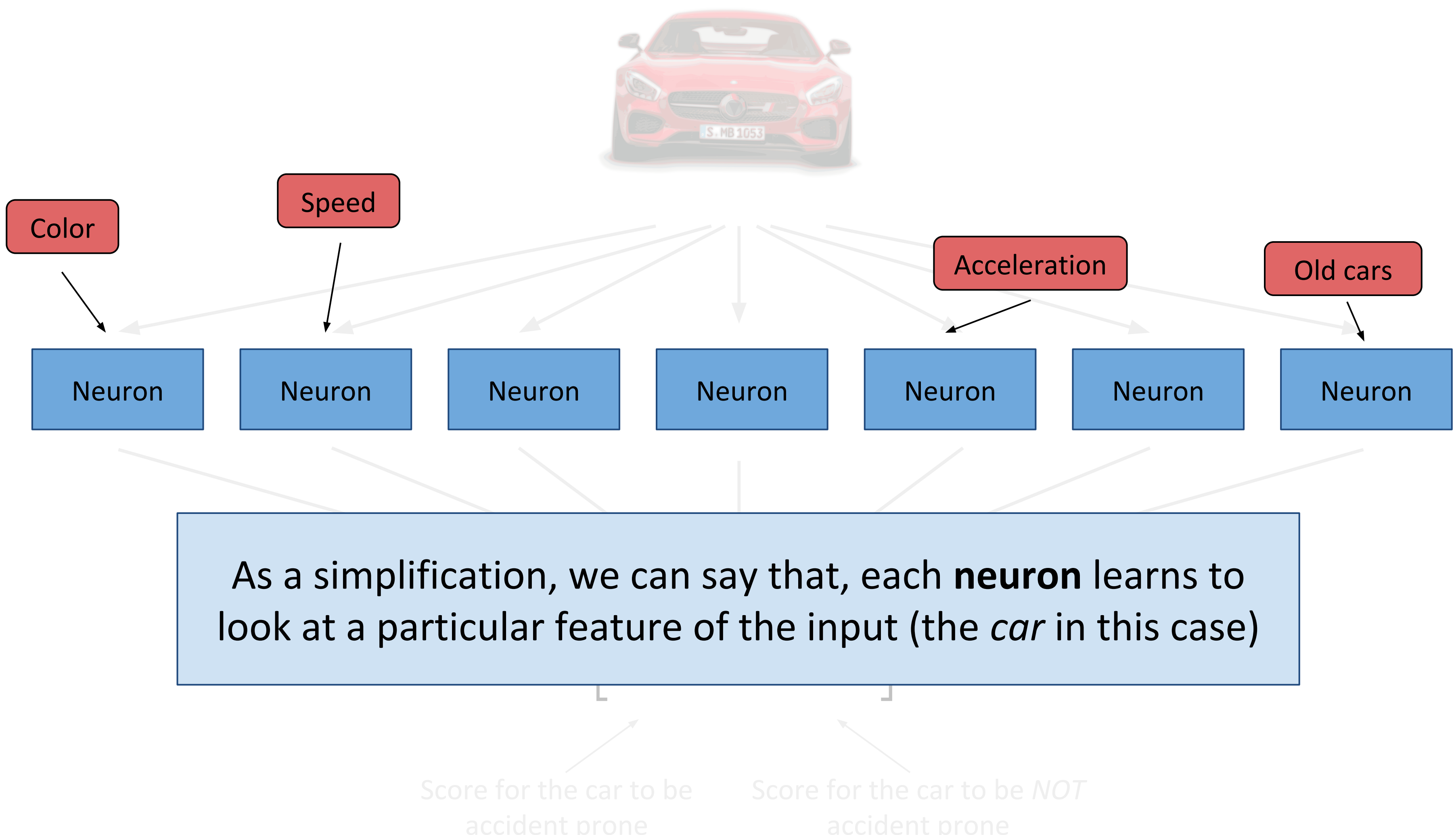
Score for the car to be *NOT* accident prone

Neural Network



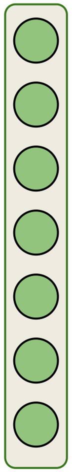
As a simplification, we can say that, each classifier learns to look at a particular feature of the input (the *car* in this case)

Neural Network

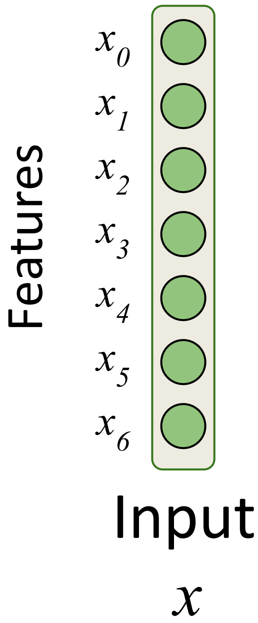


As a simplification, we can say that, each **neuron** learns to look at a particular feature of the input (the *car* in this case)

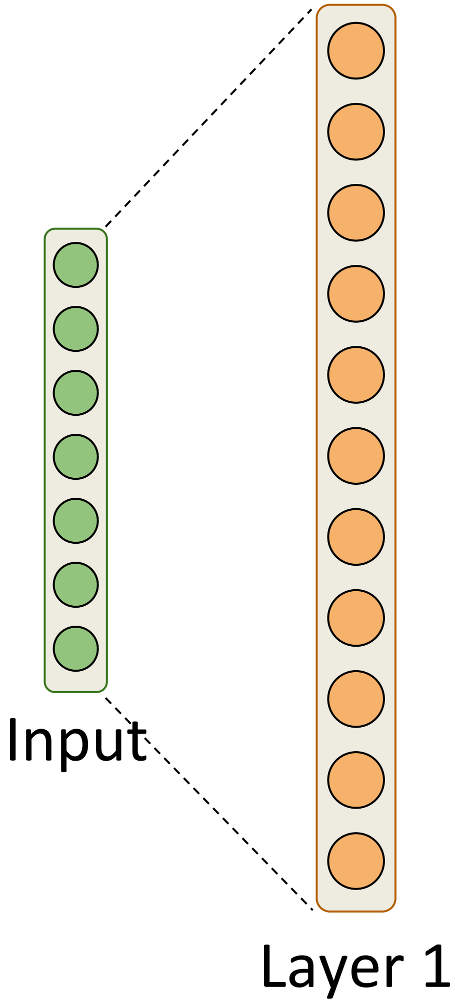
Neural Network



Neural Network

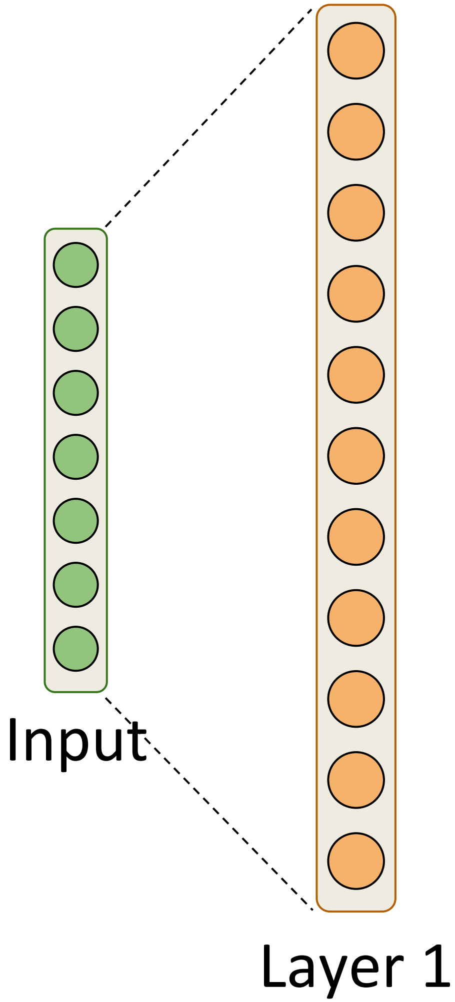


Neural Network



The neurons in the layer can be thought of as representing *richer features*

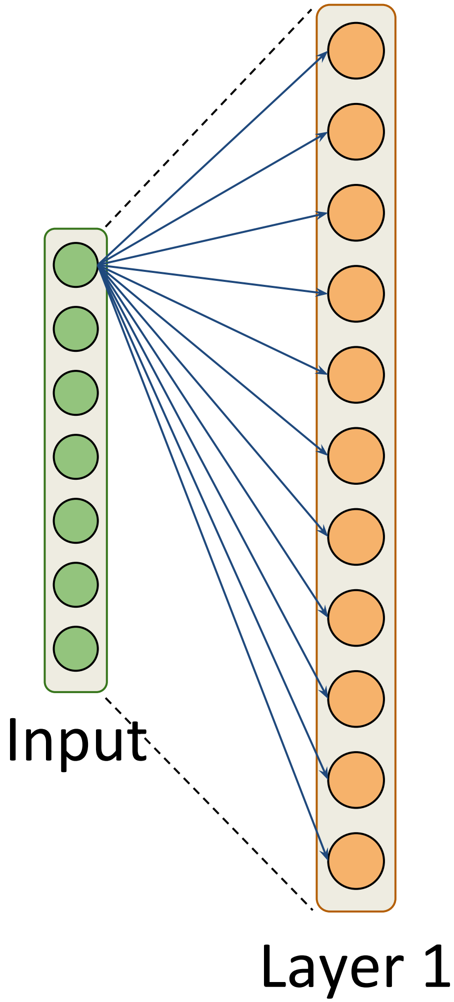
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

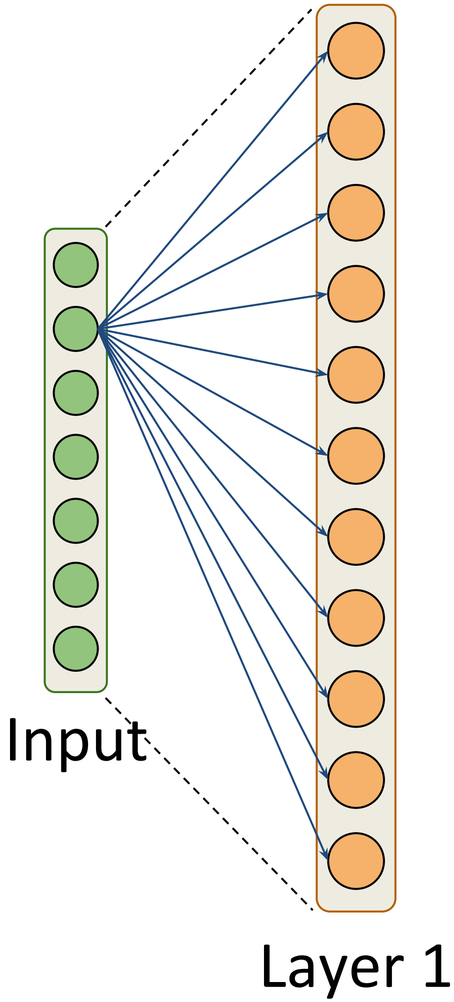
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

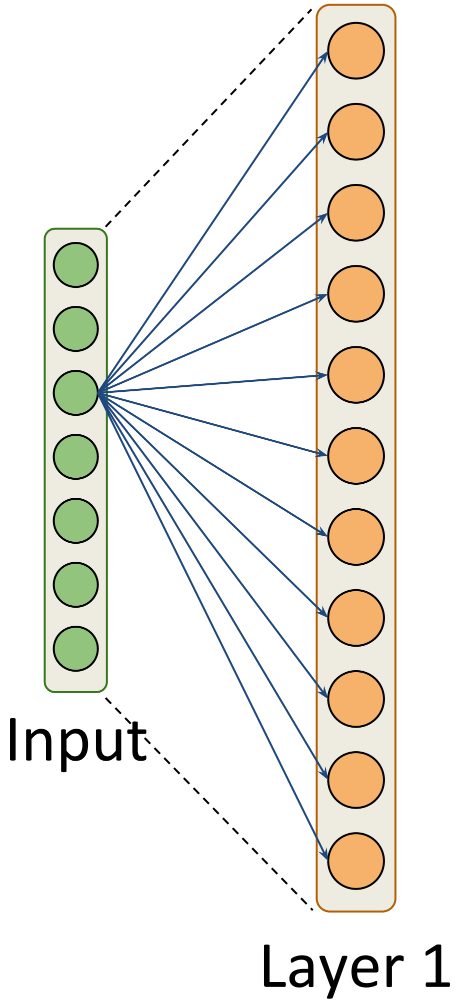
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

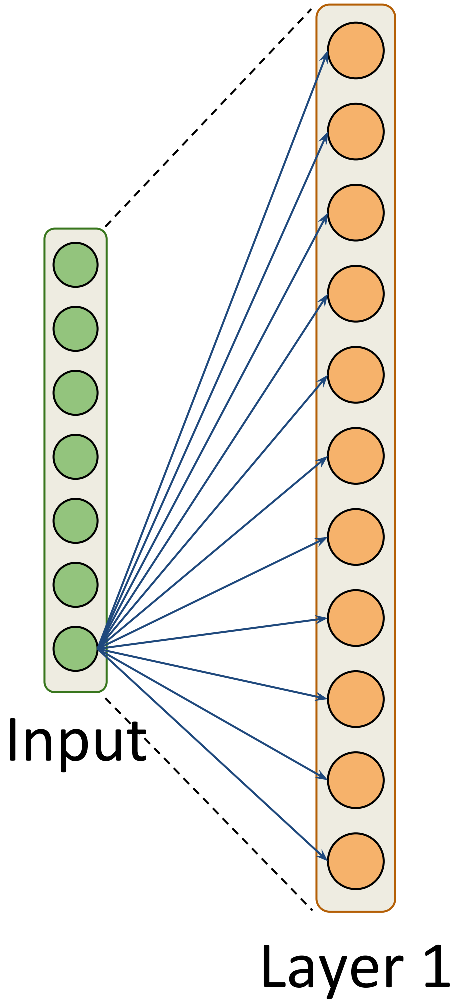
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

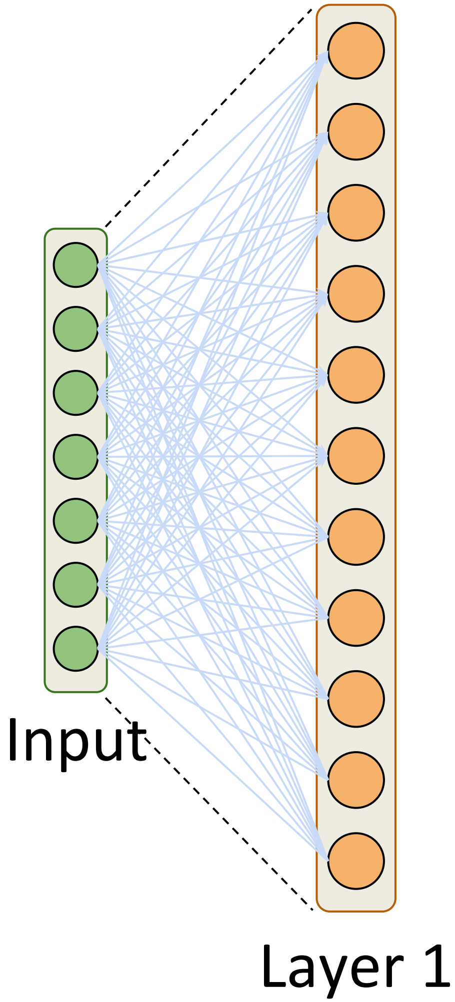
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

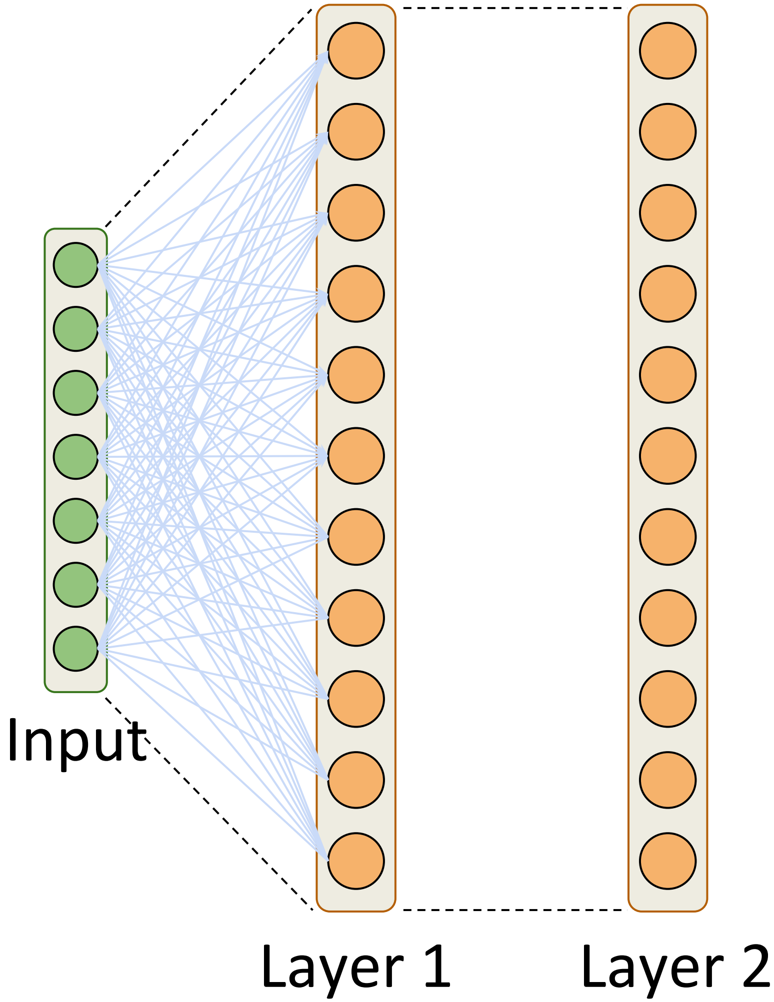
Neural Network



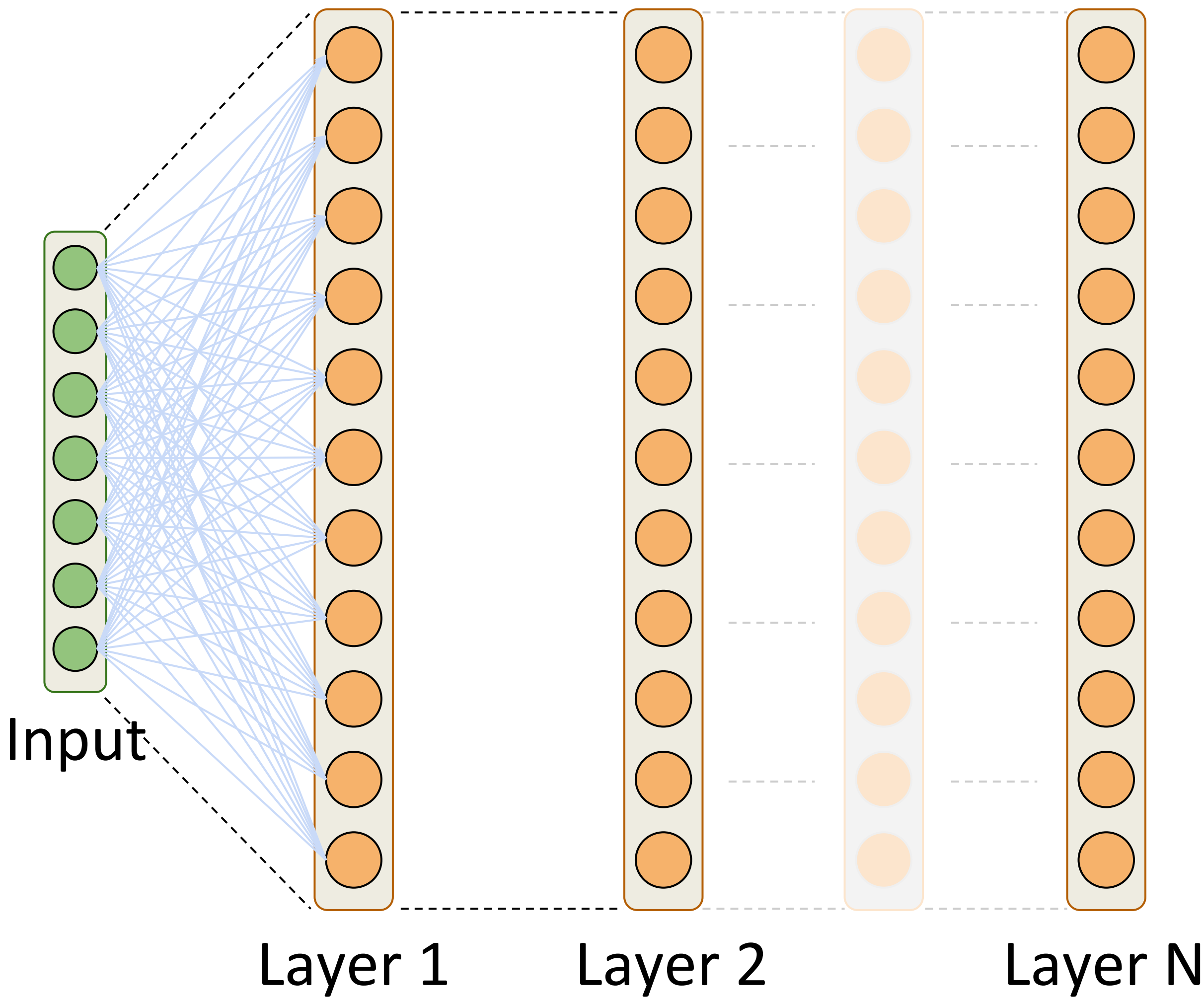
The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

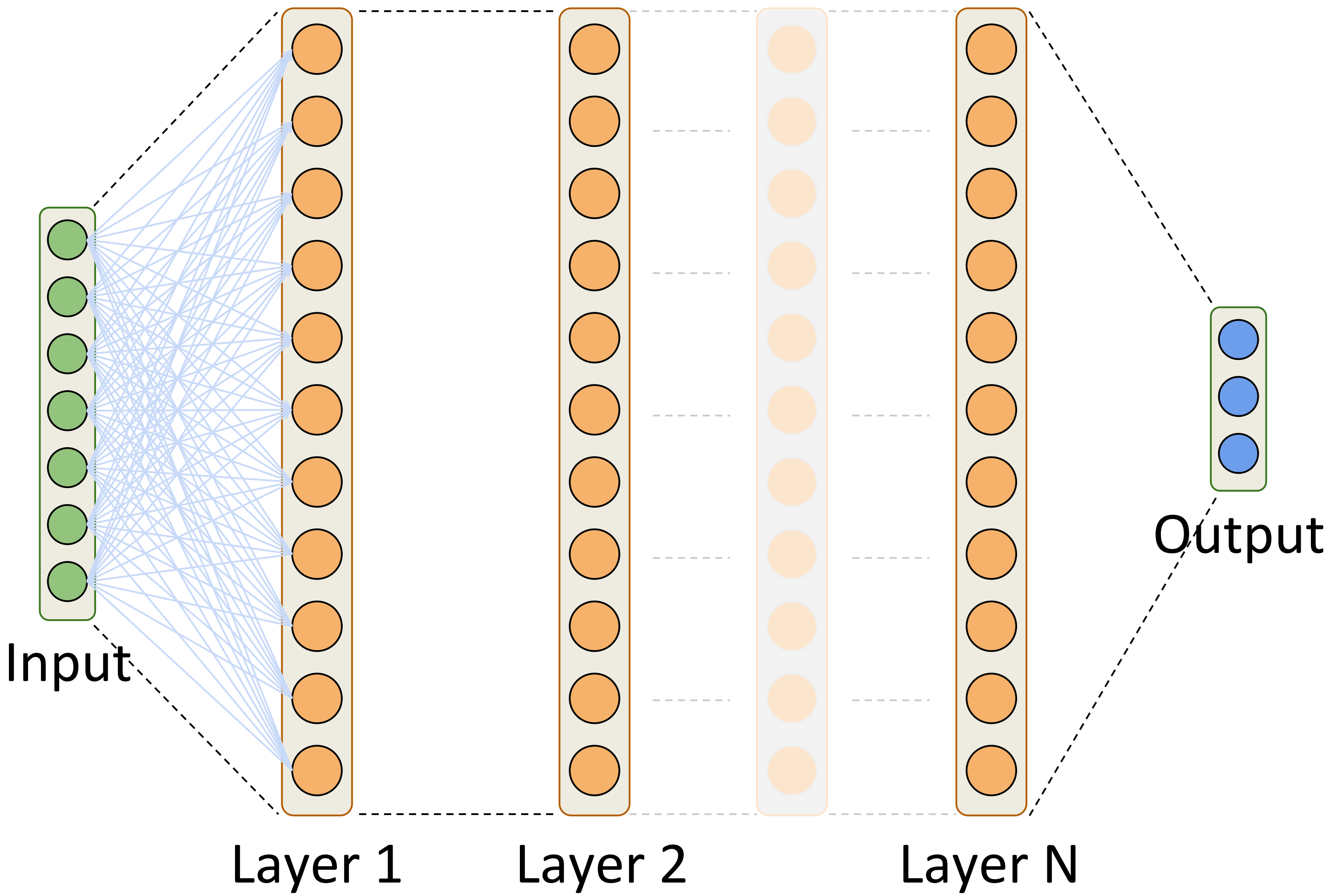
Neural Network



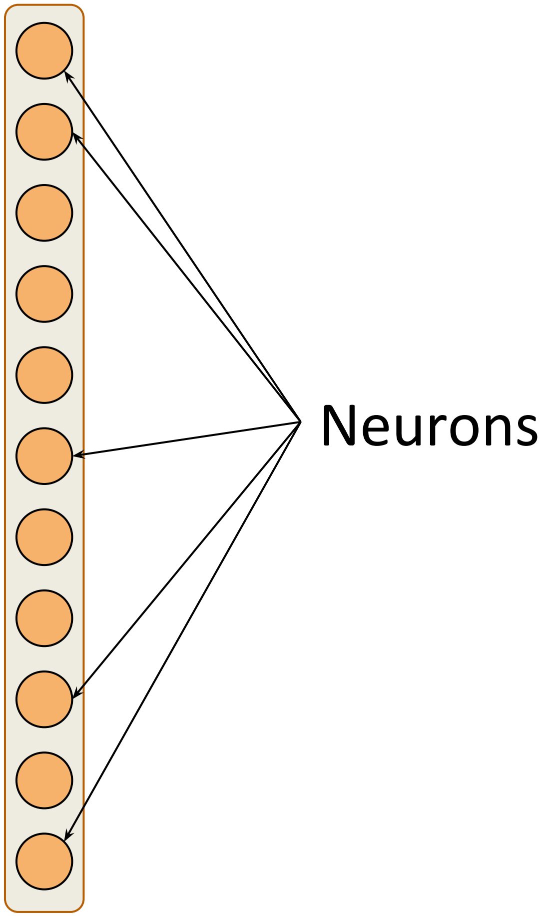
Neural Network



Neural Network

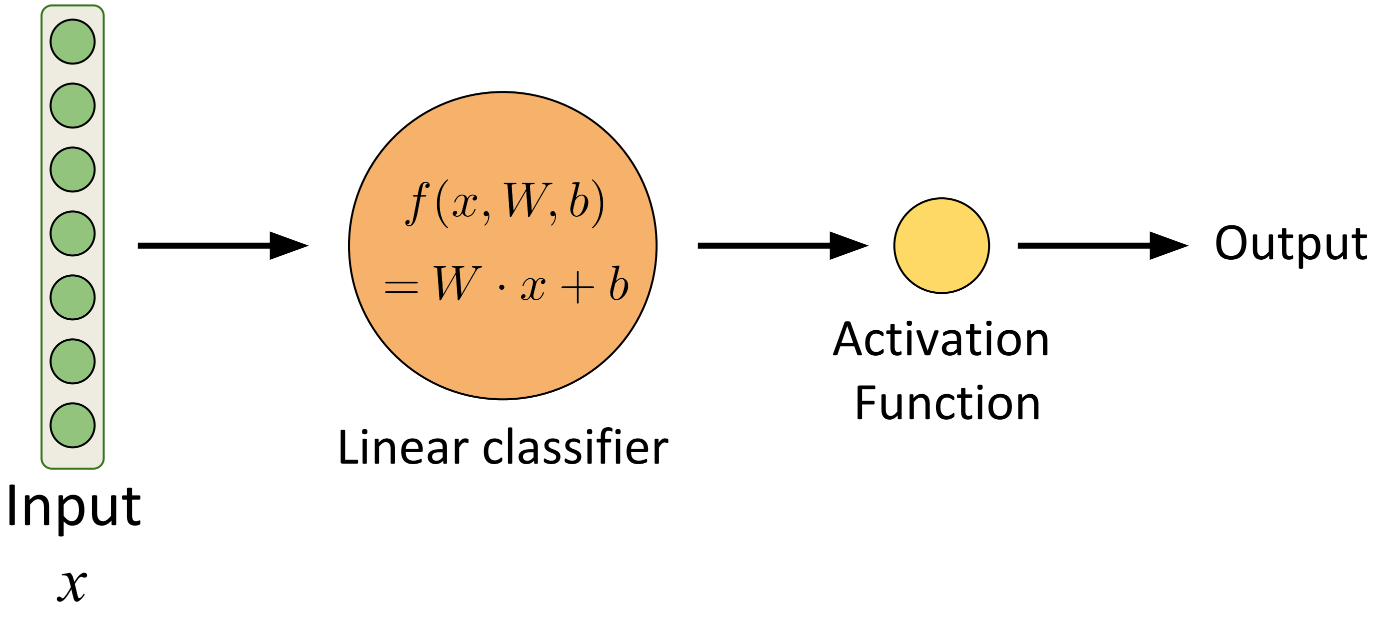


Neural Network



Neuron

A Neuron can be thought of as *a linear classifier* plus an *activation function*



Activation Functions

- Intuitively, a neuron looks at a particular feature of the data

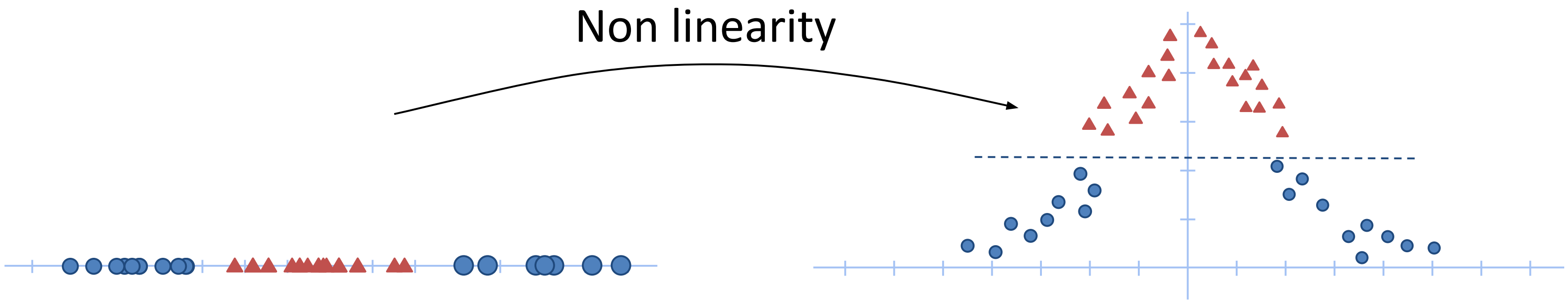
Activation Functions

- Intuitively, a neuron looks at a particular feature of the data
- The activation after the linear classifier gives us an idea of how much the neuron “supports” the feature

As an example, the output of a neuron will be high if the feature it supports is contained in the input (like “low speed” in the current “car”)

Activation Functions

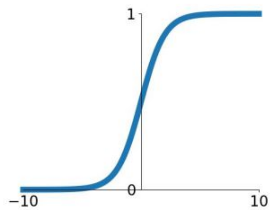
- Intuitively, a neuron looks at a particular feature of the data
- The activation after the linear classifier gives us an idea of how much the neuron “supports” the feature
- Activations also helps us map linear spaces into non-linear spaces



Activation Functions

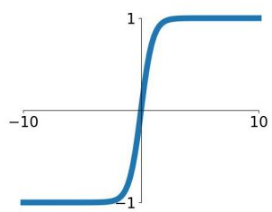
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



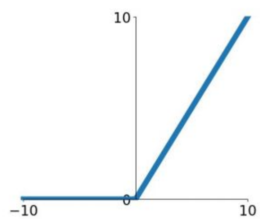
tanh

$\tanh(x)$



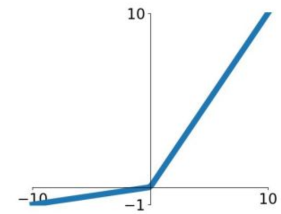
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

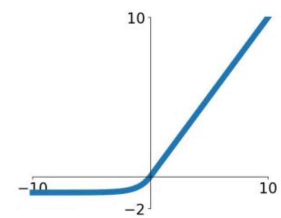


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural Network

- Entire network is nothing but a function:

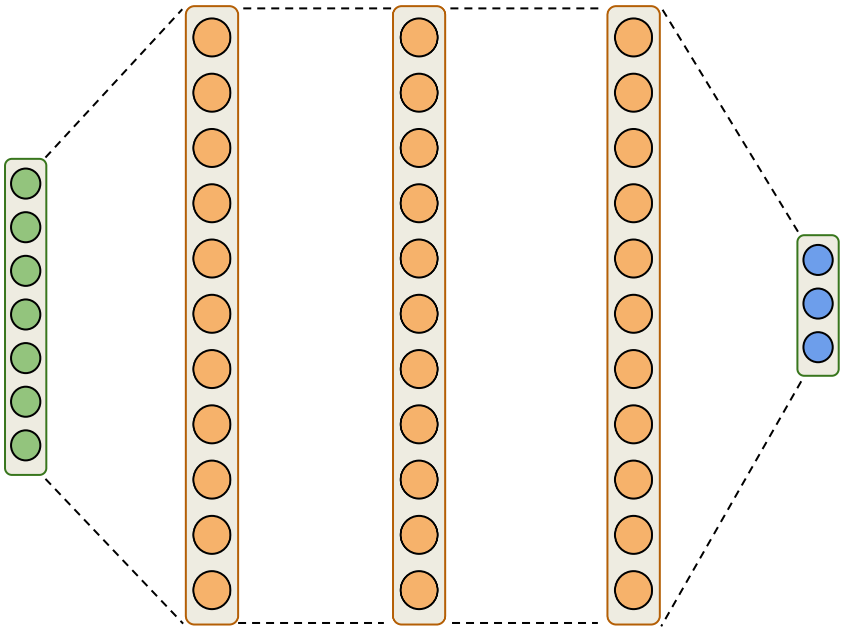
$$\boxed{f = W \cdot x + b}$$

Linear classifier

Neural Network

- Entire network is nothing but a function:

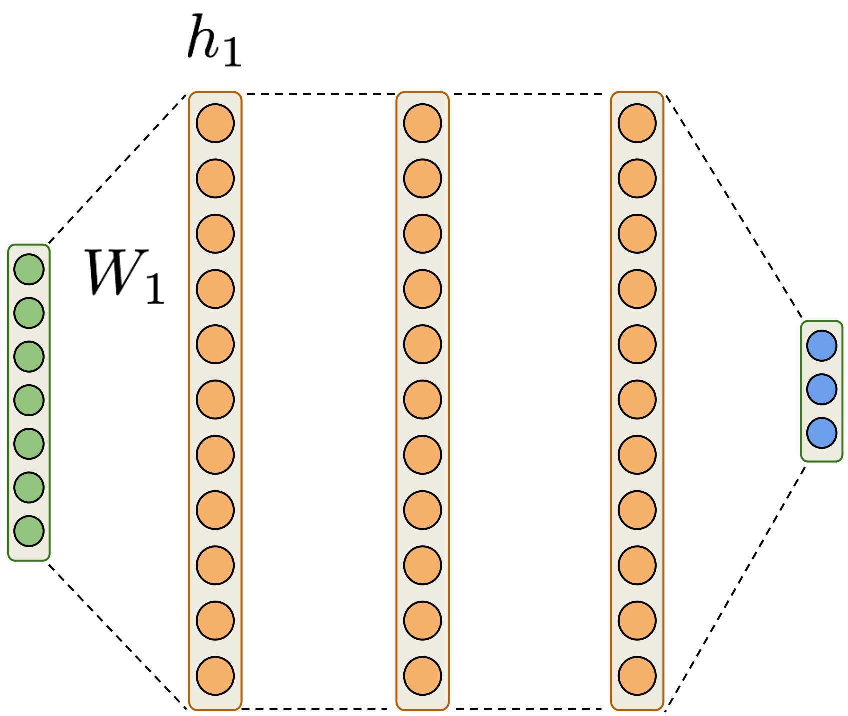
Neural network with 3 hidden layers



Neural Network

- Entire network is nothing but a function:

Neural network with 3 hidden layers

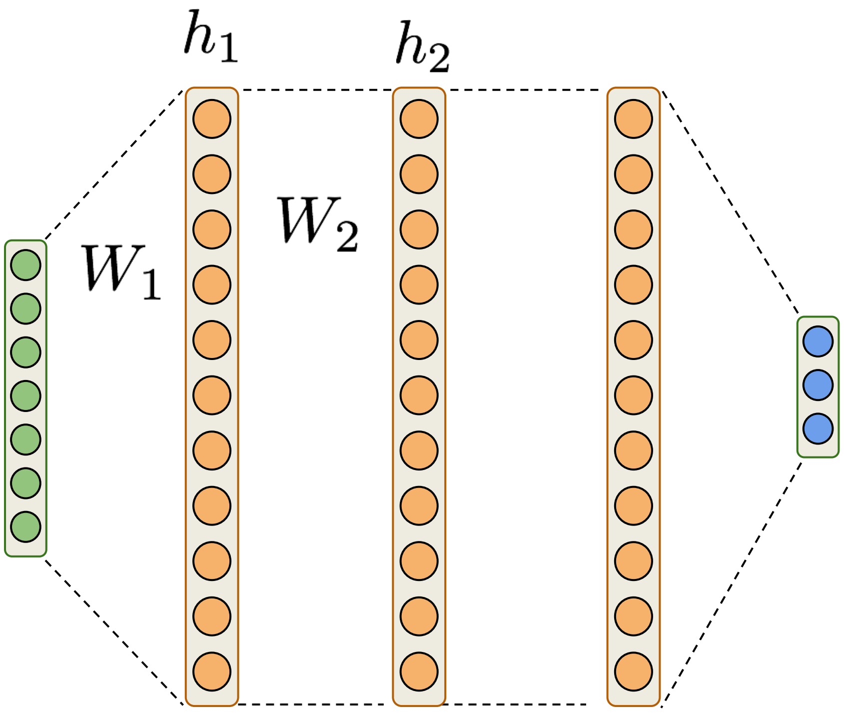


$$h_1 = \sigma(W_1 \cdot x + b_1)$$

Neural Network

- Entire network is nothing but a function:

Neural network with 3 hidden layers

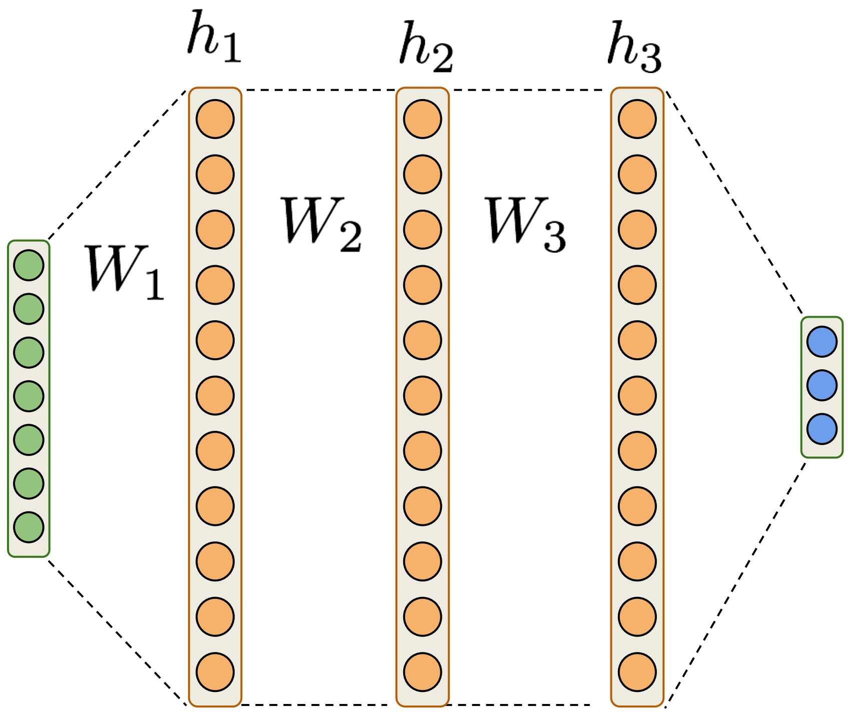


$$h_1 = \sigma(W_1 \cdot x + b_1)$$
$$h_2 = \sigma(W_2 \cdot h_1 + b_2)$$

Neural Network

- Entire network is nothing but a function:

Neural network with 3 hidden layers

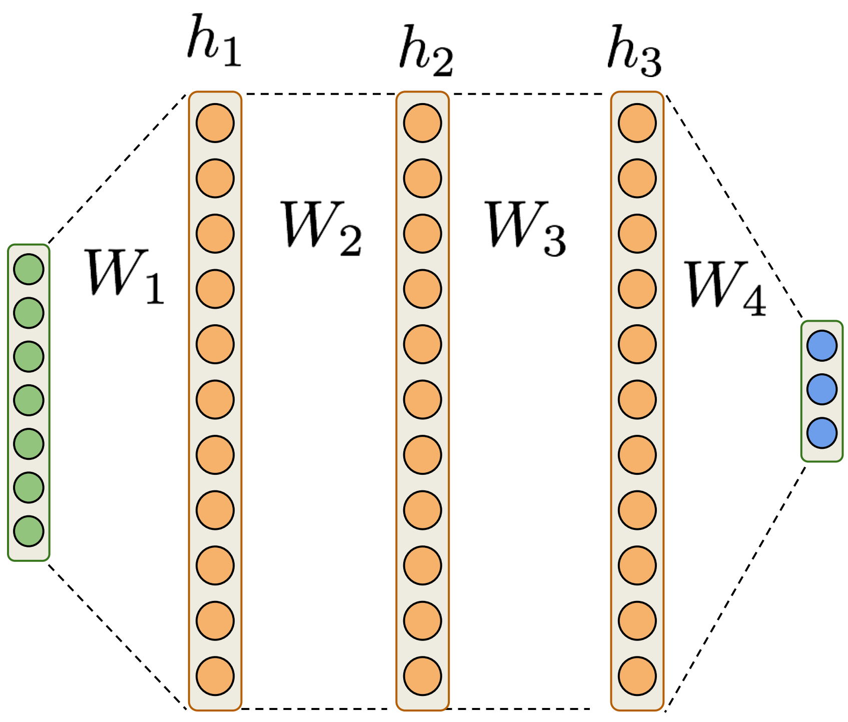


$$h_1 = \sigma(W_1 \cdot x + b_1)$$
$$h_2 = \sigma(W_2 \cdot h_1 + b_2)$$
$$h_3 = \sigma(W_3 \cdot h_2 + b_3)$$

Neural Network

- Entire network is nothing but a function:

Neural network with 3 hidden layers



$$h_1 = \sigma(W_1 \cdot x + b_1)$$

$$h_2 = \sigma(W_2 \cdot h_1 + b_2)$$

$$h_3 = \sigma(W_3 \cdot h_2 + b_3)$$

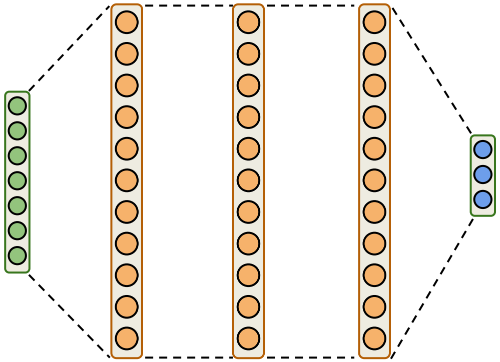
$$f = W_4 \cdot h_3 + b_4$$

Neural Network

- Everything else remains the same!

$$f = W \cdot x + b$$

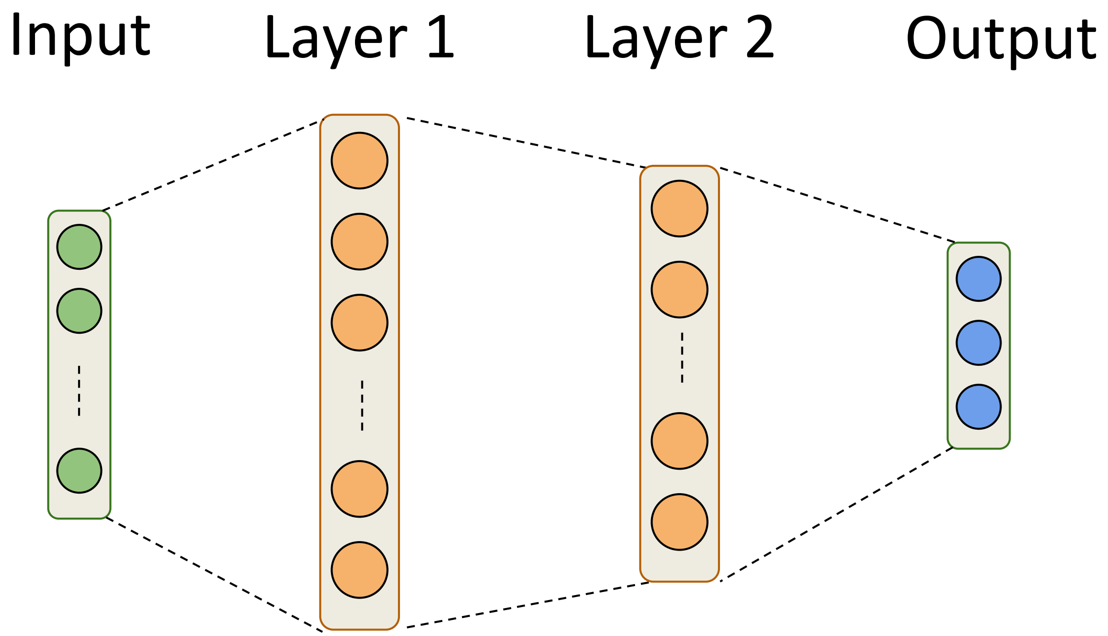
Linear classifier



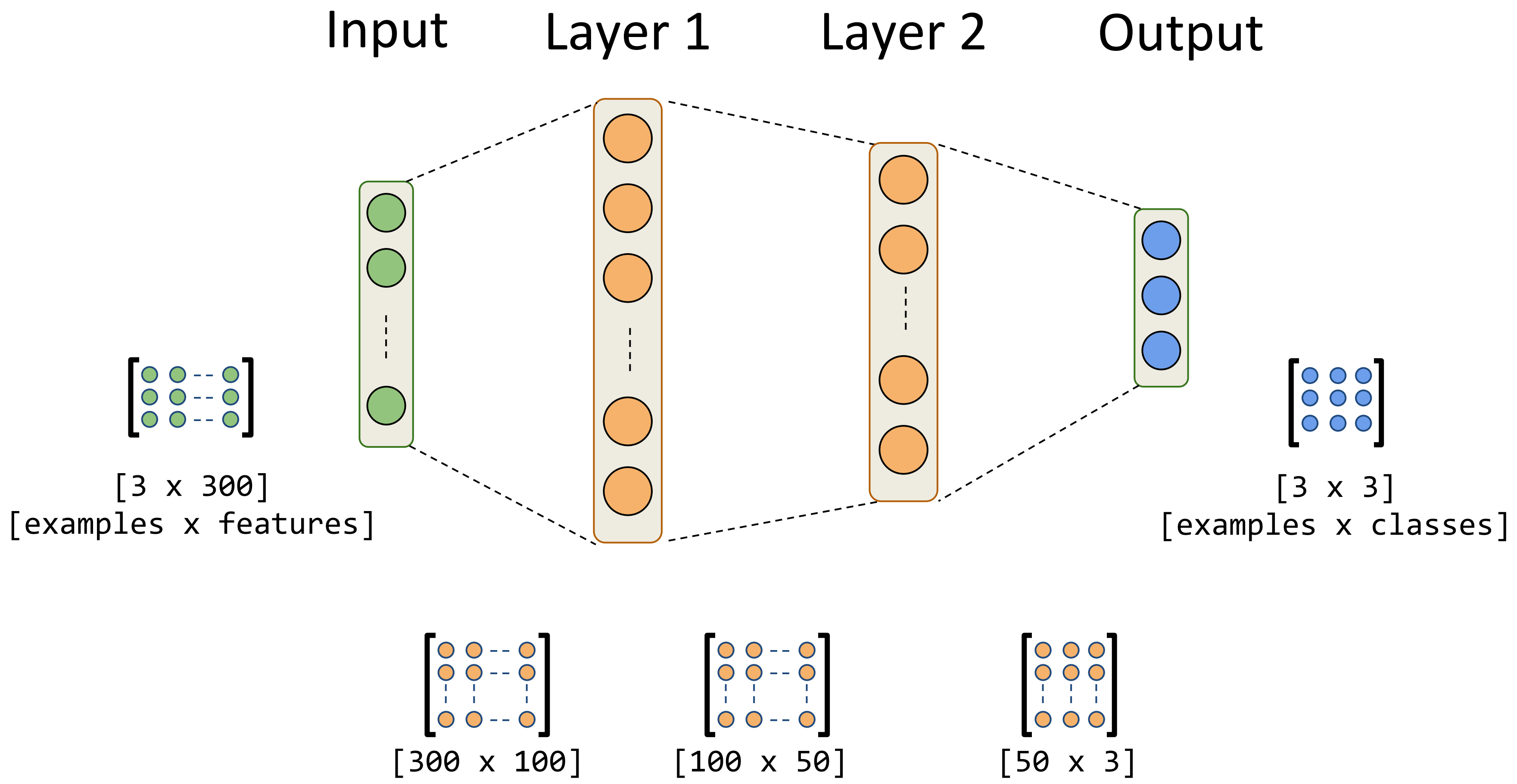
$$f = W_4 \cdot (\sigma(W_3 \cdot (\sigma(W_2 \cdot (\sigma(W_1 \cdot x + b_1)) + b_2)) + b_3)) + b_4$$

Neural network with 3 hidden layers

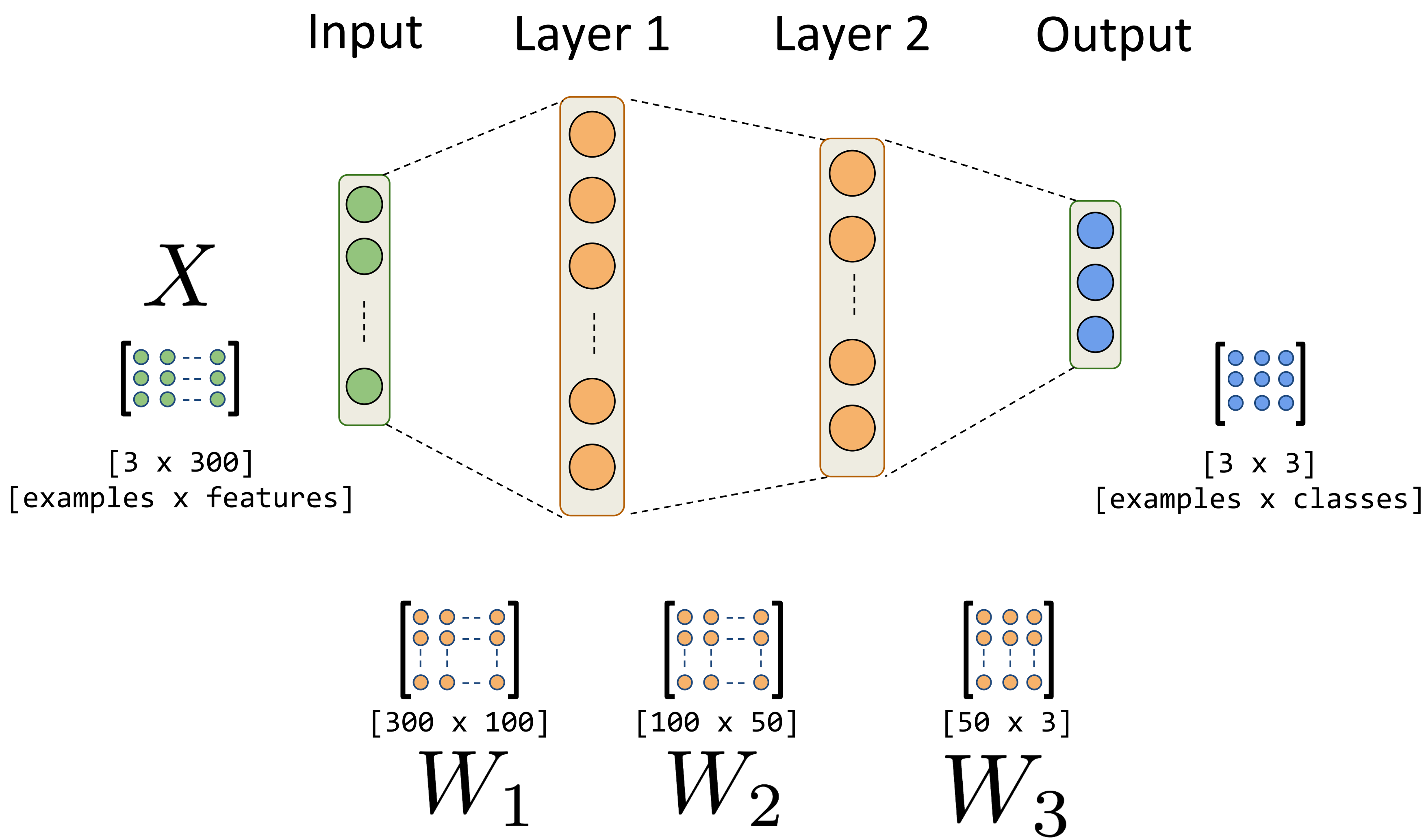
Neural Network



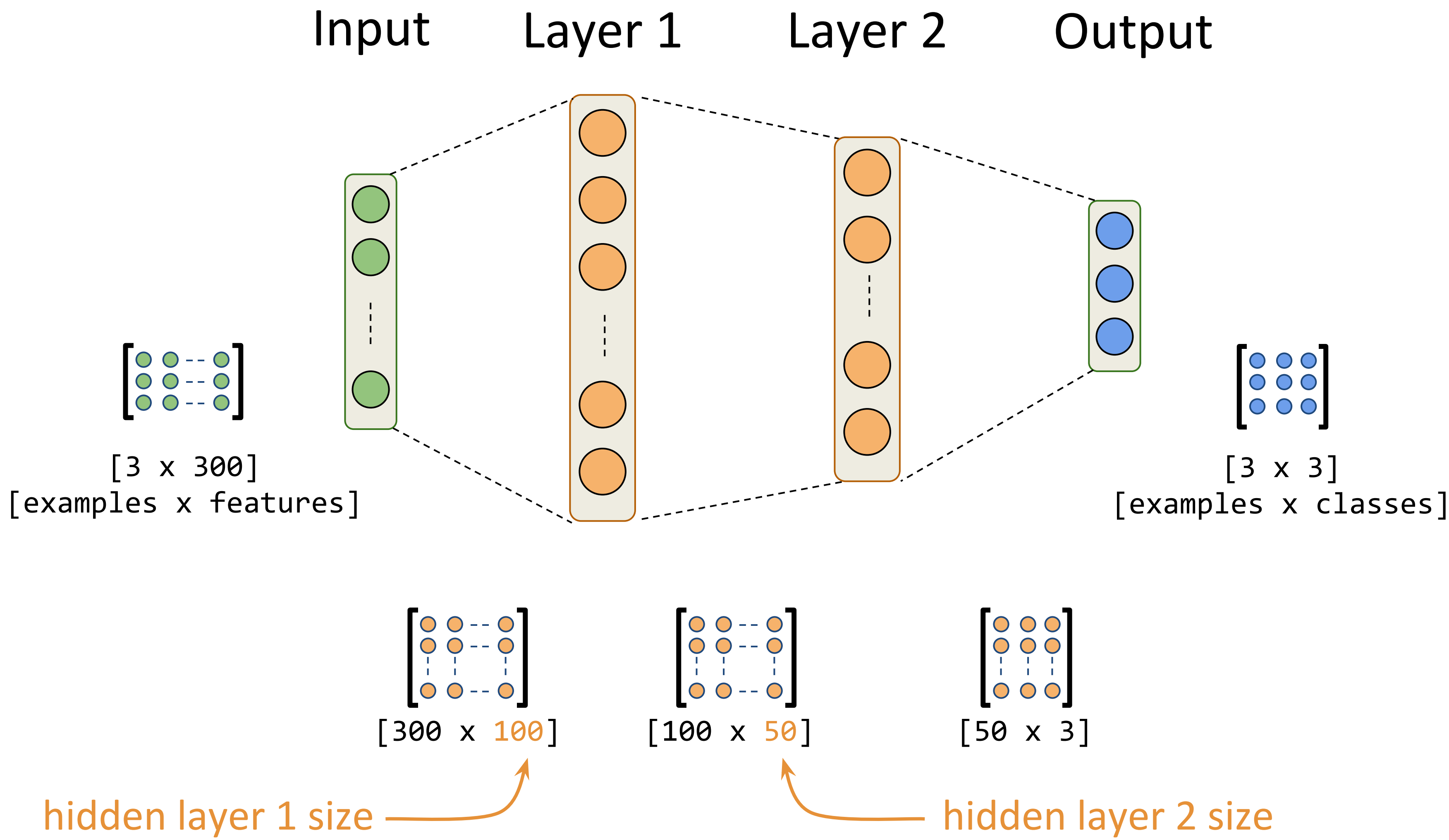
Neural Network



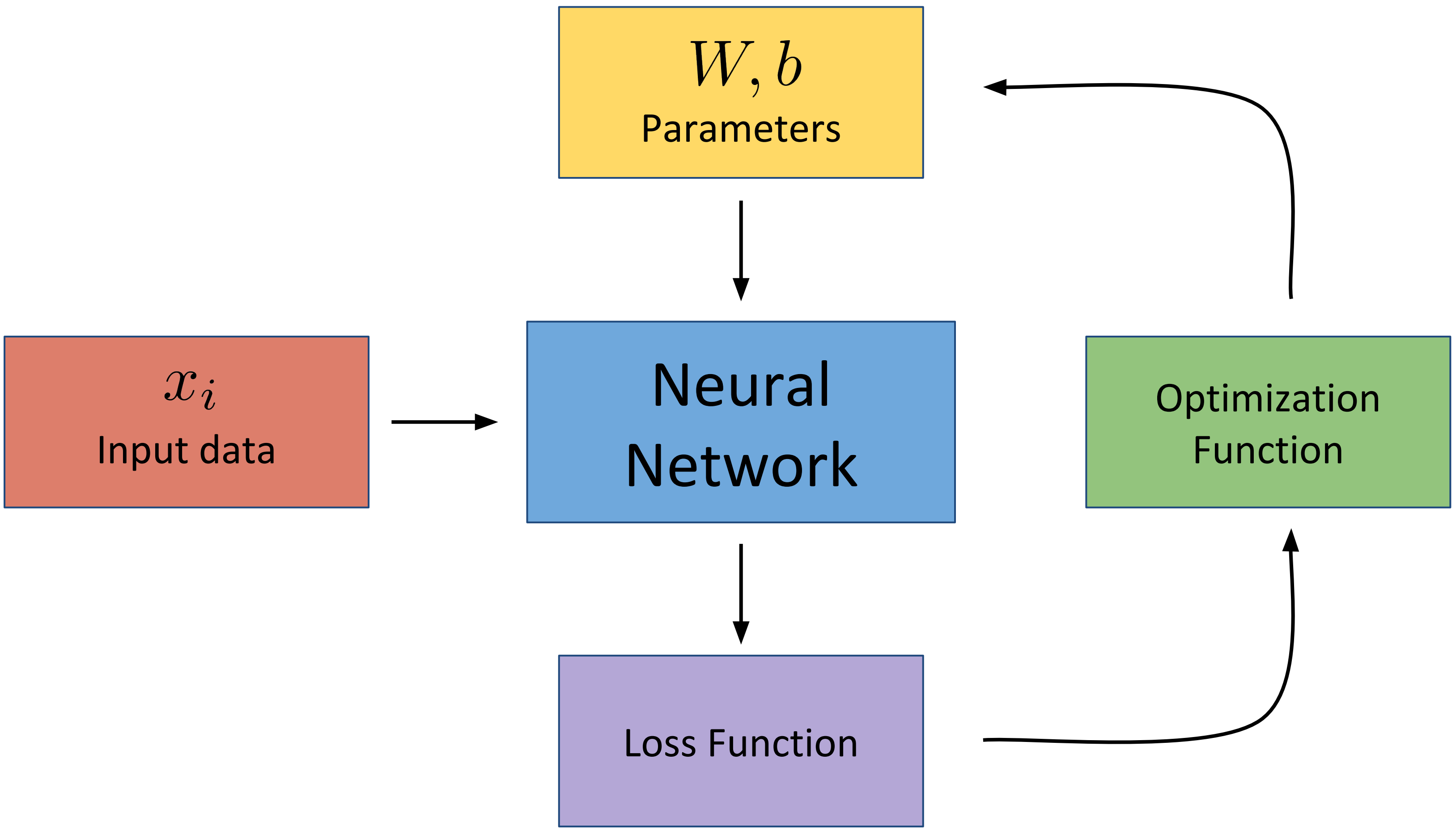
Neural Network



Neural Network



Overall Picture



Neural Network

Let’s implement a neural network model!

Neural Network

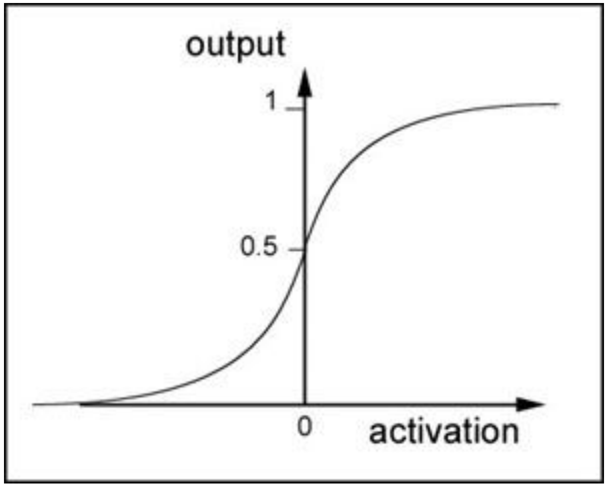
Define the parameters

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))
```

Neural Network

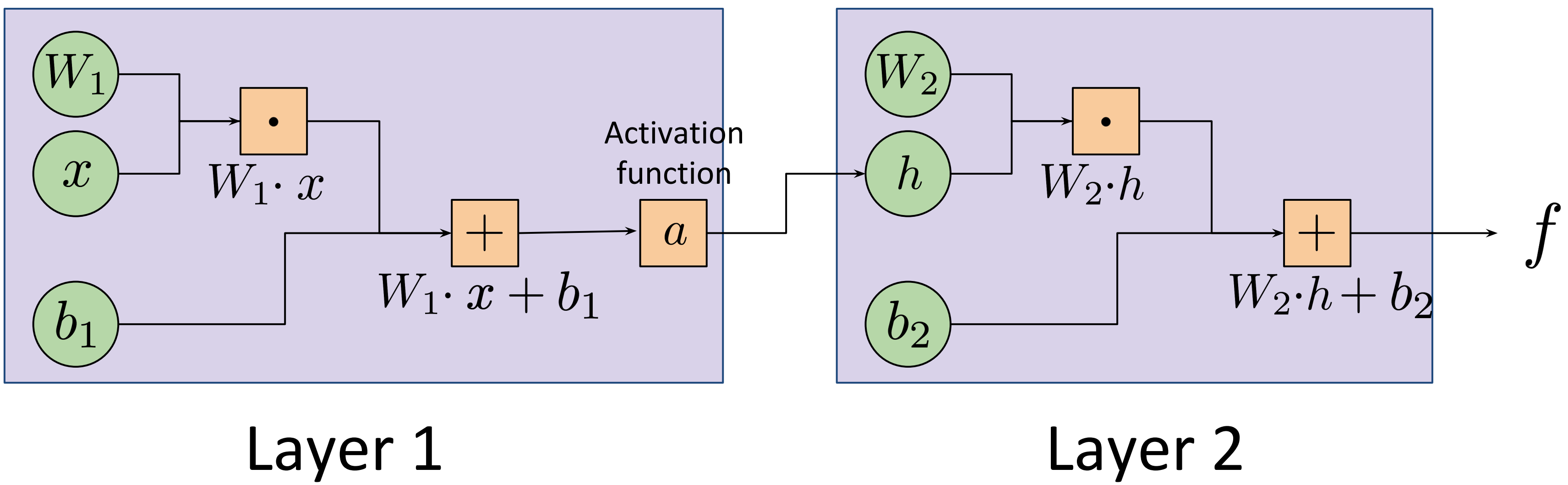
Sigmoid Activation function

$$\sigma = \frac{1}{1 + e^{-x}}$$



Neural Network

Forward Pass



Neural Network

Two level forward pass

$$h_1 = \sigma(W_1 \cdot x + b_1)$$
$$f = W_2 \cdot h_1 + b_2$$

```
# evaluate class scores, [N x K]
middle = np.dot(X, W) + b
hidden_layer = sigmoid(middle)

f = np.dot(hidden_layer, W2) + b2
```

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

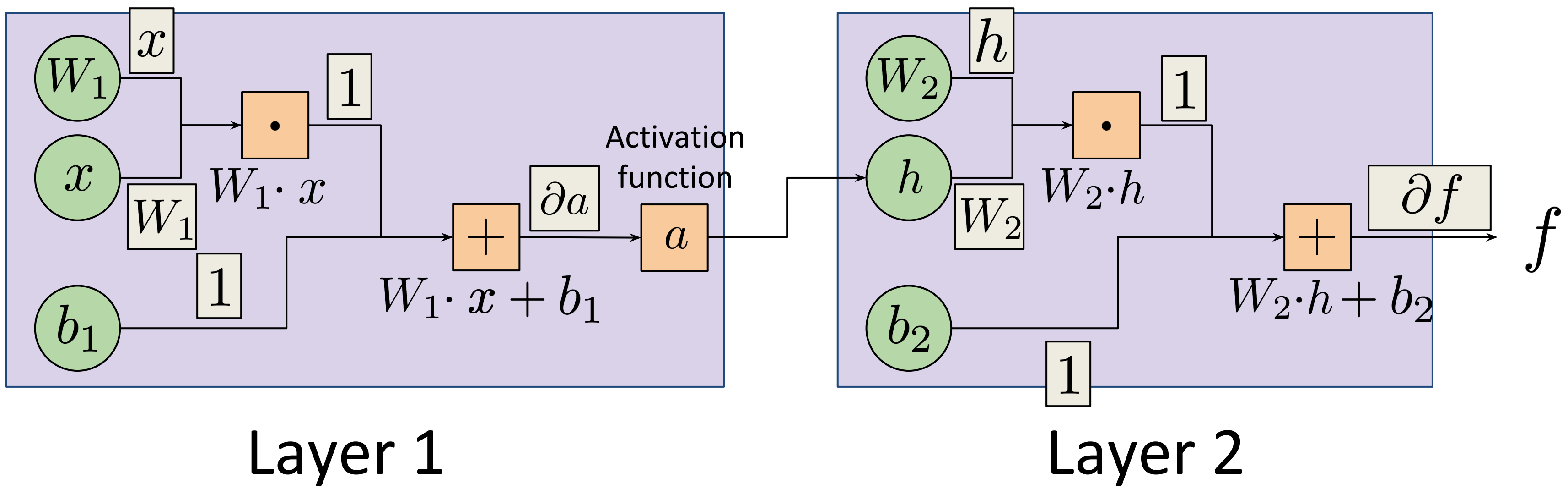
Neural Network

After forward pass calculation, everything else remains the same

- loss
- derivative of loss
- derivative propagates back to hidden layer instead of input layer

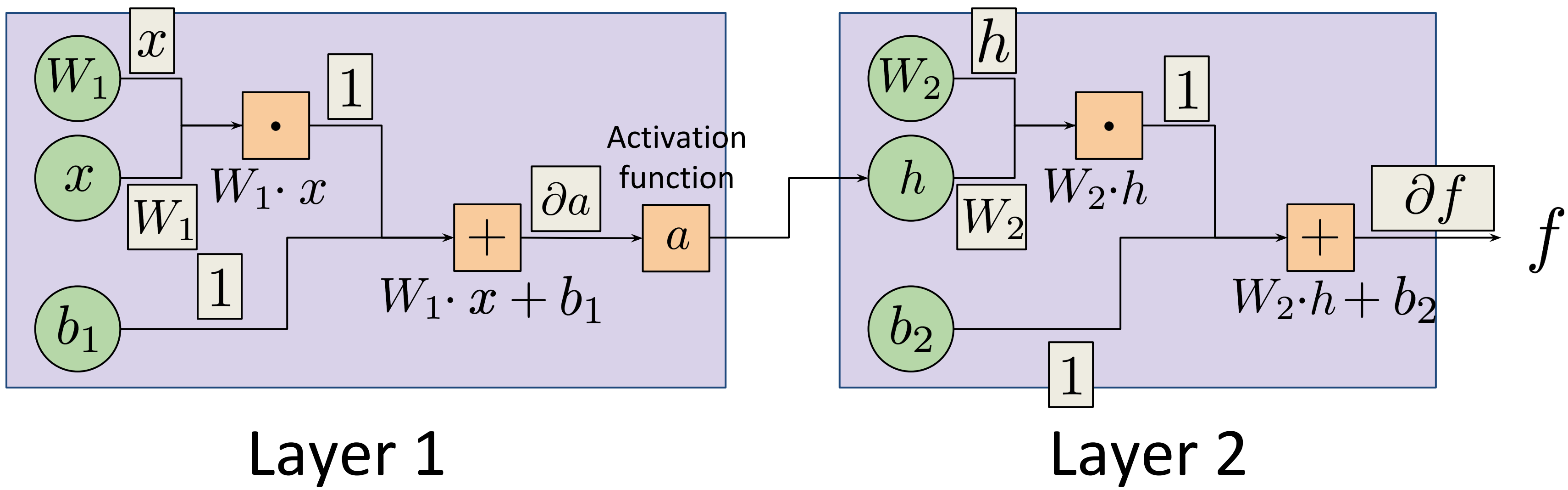
Neural Network

Forward Pass



Neural Network

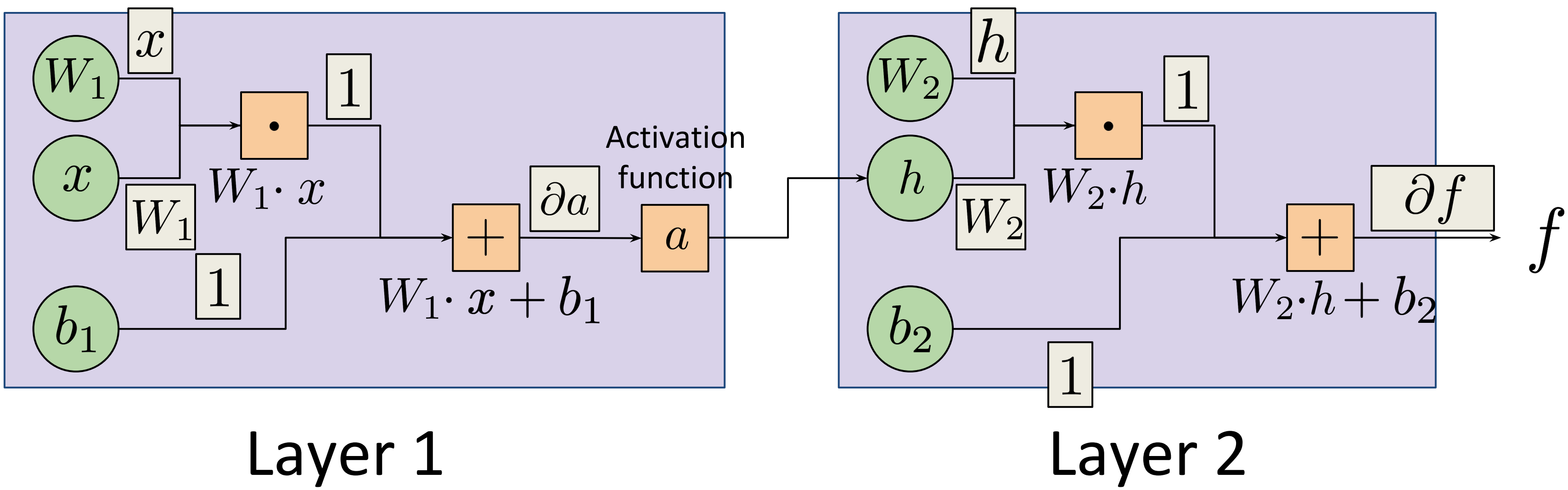
Forward Pass



$$\partial W_1 = x \cdot 1 \cdot \partial a \cdot W_2 \cdot 1 \cdot \partial f$$

Neural Network

Forward Pass

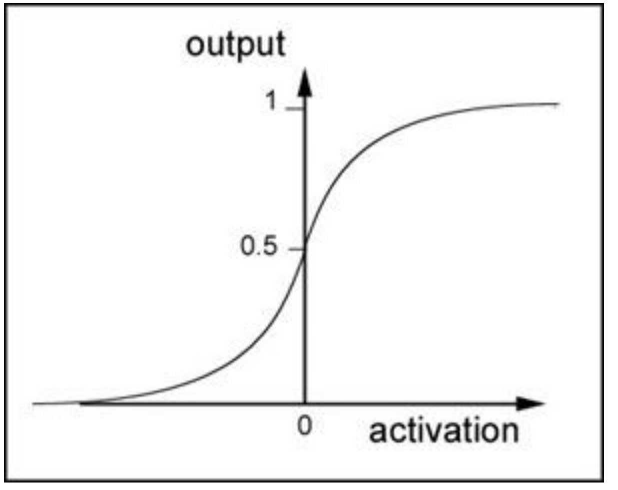


$$\partial W_2 = h \cdot 1 \cdot \partial f$$

Neural Network

Sigmoid Activation function

$$\sigma = \frac{1}{1 + e^{-x}}$$



Derivative of Sigmoid

$$\frac{\partial \sigma}{\partial x} = \sigma(1 - \sigma)$$

Neural Network

Backpropagating to hidden layer

```
# compute the gradient of function
df = probs
df[range(num_examples),y] -= 1
df /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, df)
db2 = np.sum(df, axis=0, keepdims=True)
# next backprop into hidden layer
dhidden = np.dot(df, W2.T)
```

Neural Network

Backpropagate to sigmoid function

```
# backprop the sigmoid
dhidden = dsigmoid(middle, dhidden)
```

```
def dsigmoid(x, dforward):
    t = sigmoid(x)
    return np.multiply(dforward, t * (1.0 - t))
```

Neural Network

Finally backpropagate to first layer and update the parameters

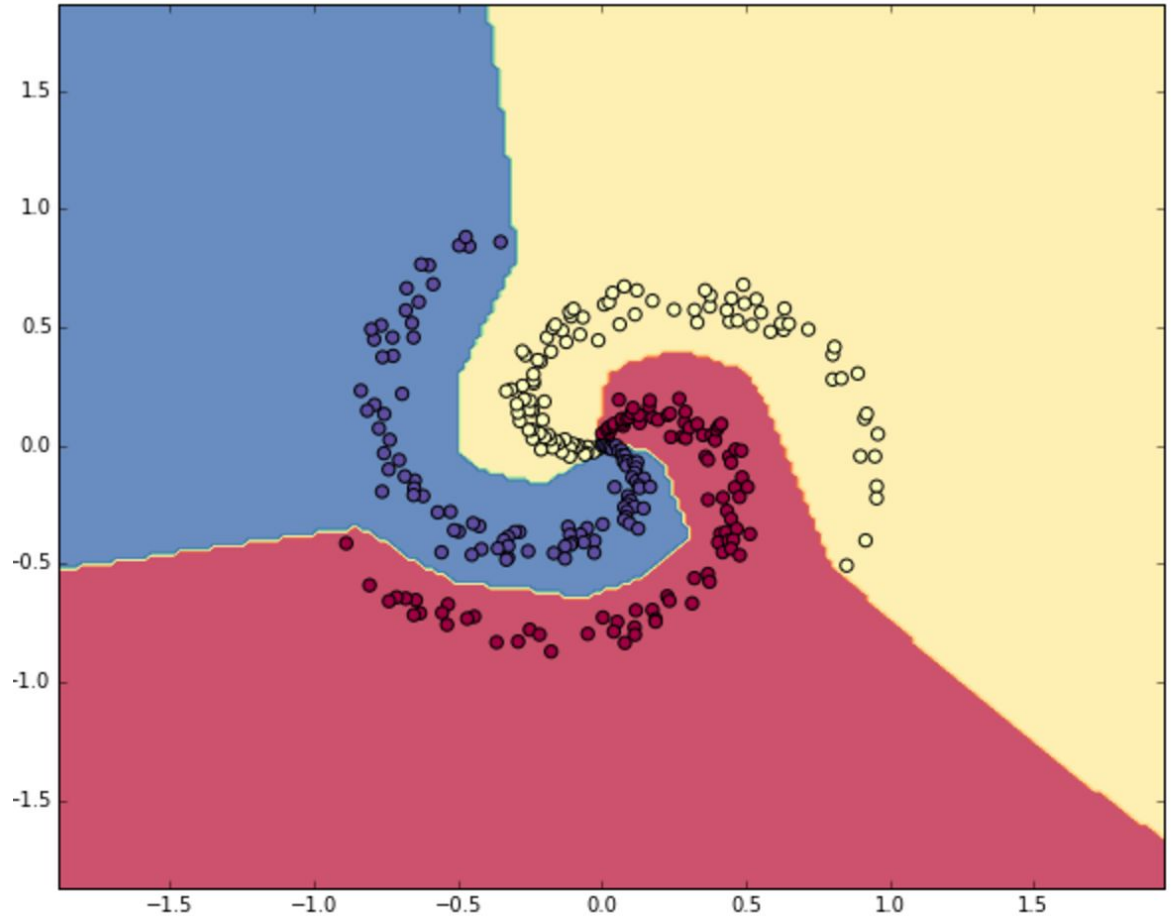
```
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)

# add regularization gradient contribution
dW2 += reg * W2
dW += reg * W

# perform a parameter update
W += -step_size * dW
b += -step_size * db
W2 += -step_size * dW2
b2 += -step_size * db2
```

Neural Network

Let’s see it in action!

Neural Network

Neural network learns the boundaries



Neural Network

Is the learning because of hidden layer or because of non-linearity added by Sigmoid?

Exercise:

1) Remove Sigmoid but keep one hidden layer and report the score
2) See the effect of learning rate

Neural Network Libraries

Gradient computation and Neural Network implementation is a lot of work!

Neural Network libraries abstract away a lot of the complexity

Neural Network Libraries

Benefits of using a library:

- Automatic differentiation
- Abstraction of neurons/layers
- Optimization/Loss functions are already implemented!

⋮

Neural Network Libraries

Many libraries to choose from:

- Theano (http://deeplearning.net/software/theano/)
- Torch (http://torch.ch)
- Tensorflow (https://www.tensorflow.org)
- MXNet (http://mxnet.io)
- Keras (https://keras.io)
- Lasagne (https://lasagne.readthedocs.io/en/latest/)
- Blocks (https://blocks.readthedocs.io/en/latest/)

⋮

Neural Network Libraries

```python
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def dsigmoid(x, dforward):
    t = sigmoid(x)
    return np.multiply(dforward, t * (1.0 - t))

# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

# some hyperparameters
step_size = 1e0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in xrange(10000):

    # evaluate class scores, [N x K]
    middle = np.dot(X, W) + b
    hidden_layer = sigmoid(middle)

    f = np.dot(hidden_layer, W2) + b2

    # compute the class probabilities
    exp_f = np.exp(f)
    probs = exp_f / np.sum(exp_f, axis=1, keepdims=True) # [N x K]

    # compute the loss: average cross-entropy loss and regularization
    corect_logprobs = -np.log(probs[range(num_examples),y])
    data_loss = np.sum(corect_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
    loss = data_loss + reg_loss
    if i % 1000 == 0:
        print "iteration %d: loss %f" % (i, loss)

    # compute the gradient of function
    df = probs
    df[range(num_examples),y] -= 1
    df /= num_examples

    # backpropate the gradient to the parameters
    # first backprop into parameters W2 and b2
    dW2 = np.dot(hidden_layer.T, df)
    db2 = np.sum(df, axis=0, keepdims=True)
    # next backprop into hidden layer
    dhidden = np.dot(df, W2.T)

    # backprop the sigmoid
    dhidden = dsigmoid(middle, dhidden)

    # finally into W,b
    dW = np.dot(X.T, dhidden)
    db = np.sum(dhidden, axis=0, keepdims=True)

    # add regularization gradient contribution
    dW2 += reg * W2
    dW += reg * W

    # perform a parameter update
    W += -step_size * dW
    b += -step_size * db
    W2 += -step_size * dW2
    b2 += -step_size * db2
```

```python
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(K, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=False, epochs=10000)
```

Keras

Neural Network Libraries

```python
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# ... Load the data ...
num_classes = 3
num_examples = 50
X = np.random.random((num_examples,5))
y = np.random.random((num_examples,num_classes))

# ... Build the model ...
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=True, epochs=100)
```

Neural Network Libraries

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np
```

Import modules

```
# ... Load the data ...
num_classes = 3
num_examples = 50
X = np.random.random((num_examples,5))
y = np.random.random((num_examples,num_classes))

# ... Build the model ...
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=True, epochs=100)
```

Neural Network Libraries

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# ... Load the data ...
num_classes = 3
num_examples = 50
X = np.random.random((num_examples,5))
y = np.random.random((num_examples,num_classes))

# ... Build the model ...
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=True, epochs=100)
```

Data loading

Neural Network Libraries

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# ... Load the data ...
num_classes = 3
num_examples = 50
X = np.random.random((num_examples,5))
y = np.random.random((num_examples,num_classes))

# ... Build the model ...
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=True, epochs=100)
```

Model building

One hidden layer

ReLU activation

Softmax on output layer

Neural Network Libraries

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# ... Load the data ...
num_classes = 3
num_examples = 50
X = np.random.random((num_examples,5))
y = np.random.random((num_examples,num_classes))

# ... Build the model ...
model = Sequential()
model.add(Dense(100, input_shape=(len(X[0]),), activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Setup optimization/loss

```
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(X, y, verbose=True, epochs=100)
```

SGD optimizer

Cross Entropy Loss

Neural Network Exercise

Let’s work on real world data using Keras

1) Report accuracy with 1, 2 and 3 hidden layers of size 100
2) Report accuracy with 2 hidden layers of sizes 100, 200 and 300
3) Report accuracy with Sigmoid vs ReLU activations between the hidden layers
4) Find the best hyper-parameters!