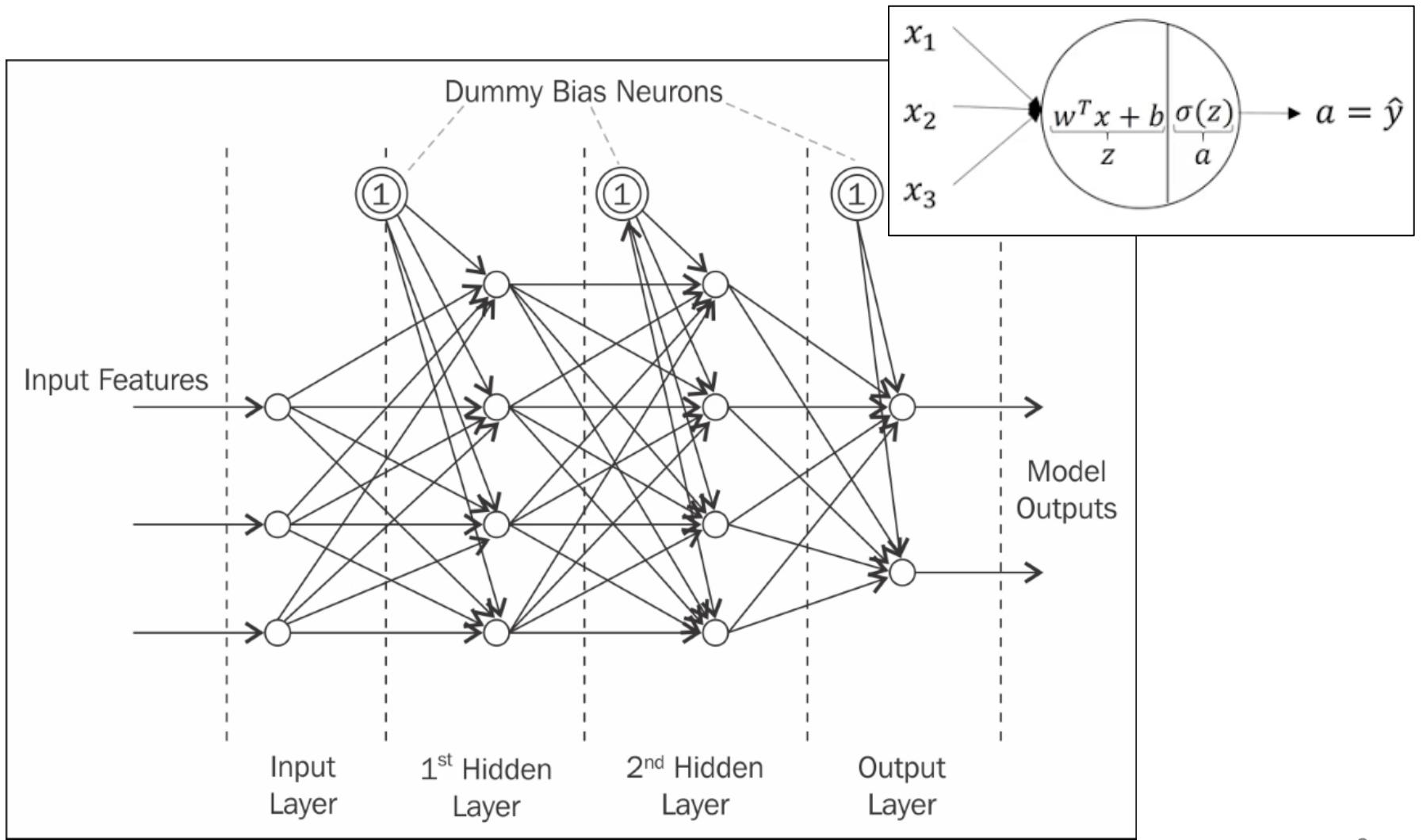


# Processamento e Análise de Imagens

## The Multilayer Perceptron

Prof. Alexei Machado  
PUC Minas

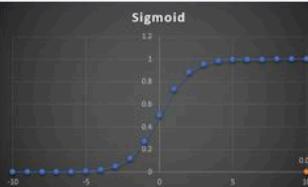
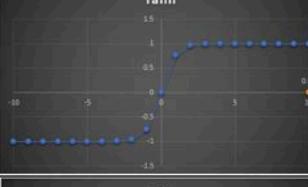
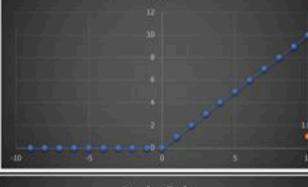
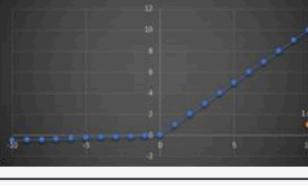
# The Multi-Layer Perceptron



# Notation

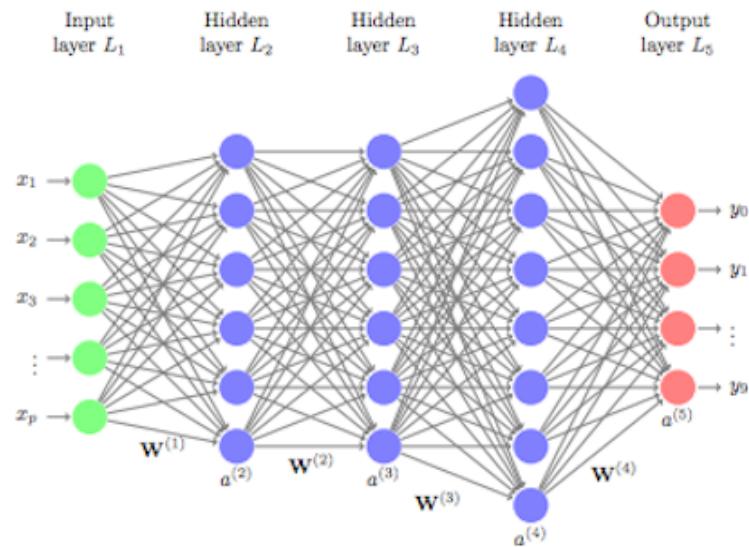
- $X$  is the  $n^{[0]}, m$  data matrix of  $m$  samples
- $Y$  is the  $n^{[L]}, m$  label matrix of  $m$  samples
- $[l]$  is the layer index,  $l=0 \dots L$
- $L$  is the number of layers
- $x^{(i)}$  denotes the  $i$ -th sample
- $x^{(i)}_j$  denotes the  $j$ -th variable of the  $i$ -th sample
- Each layer  $l$  has  $n^{[l]}$  units
- $W^{[l]}$  is the  $n^{[l]}, n^{[l-1]}$  weight matrix of layer  $l$
- $b^{[l]}$  is the  $n^{[l]}$  bias array in layer  $l$
- $Z^{[l]}$  is the  $n^{[l]}, m$  matrix with  $W^{[l]}A^{[l-1]}+b^{[l]}$
- $A^{[l]}$  is the  $n^{[l]}, m$  matrix with  $g(Z^{[l]})$
- $X=A^{[0]}; Y^{\wedge}=A^{[L]}$

# Activation Functions

Name	Plot	Equation	Derivative
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$
Rectified Linear Unit (relu)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Leaky Rectified Linear Unit (Leaky relu)		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

# Deep MLP

- A shallow network can compute any function, but with an exponential number of hidden units
- Using more hidden layers decreases the number of units



[http://uc-r.github.io/feedforward\\_DNN](http://uc-r.github.io/feedforward_DNN)

# Deep MLP Algorithm

1. Initialize parameters / Define hyperparameters
2. Loop for num\_iterations:
  - a. Forward propagation
  - b. Compute cost function
  - c. Backward propagation
  - d. Update parameters (using parameters, and grads from backprop)
4. Use trained parameters to predict labels

# MLP Gradient Descent – Forward/Loss

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[\ell]} = W^{[\ell]}A^{[\ell-1]} + b^{[\ell]}$$

$$A^{[\ell]} = g^{[\ell]}(Z^{[\ell]})$$

$$\mathcal{J}(W, b) = -\frac{1}{m} \sum (\gamma \ln \hat{y} + (1 - \gamma) \ln(1 - \hat{y}))$$

# MLP Gradient Descent – Backward/ Update

$$dA^{[L]} = -Y / A^{[L]} - (1 - Y) / (1 - A^{[L]})$$

$$dZ^{[L]} = A^{[L]} - Y$$

$$dA^{[L-1]} = W^{[L]T} dZ^{[L]}$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}$$

$$db^{[L]} = \frac{1}{m} \sum dZ^{[L]}$$

$$dZ^{[1]} = dA^{[1]} * g^{[1]'}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} \sum dZ^{[1]}$$

$$dZ^{[J]} = dA^{[J]} * g^{[J]'}(Z^{[J]})$$

$$dA^{[J-1]} = W^{[J]T} dZ^{[J]}$$

$$dW^{[J]} = \frac{1}{m} dZ^{[J]} A^{[J-1]T}$$

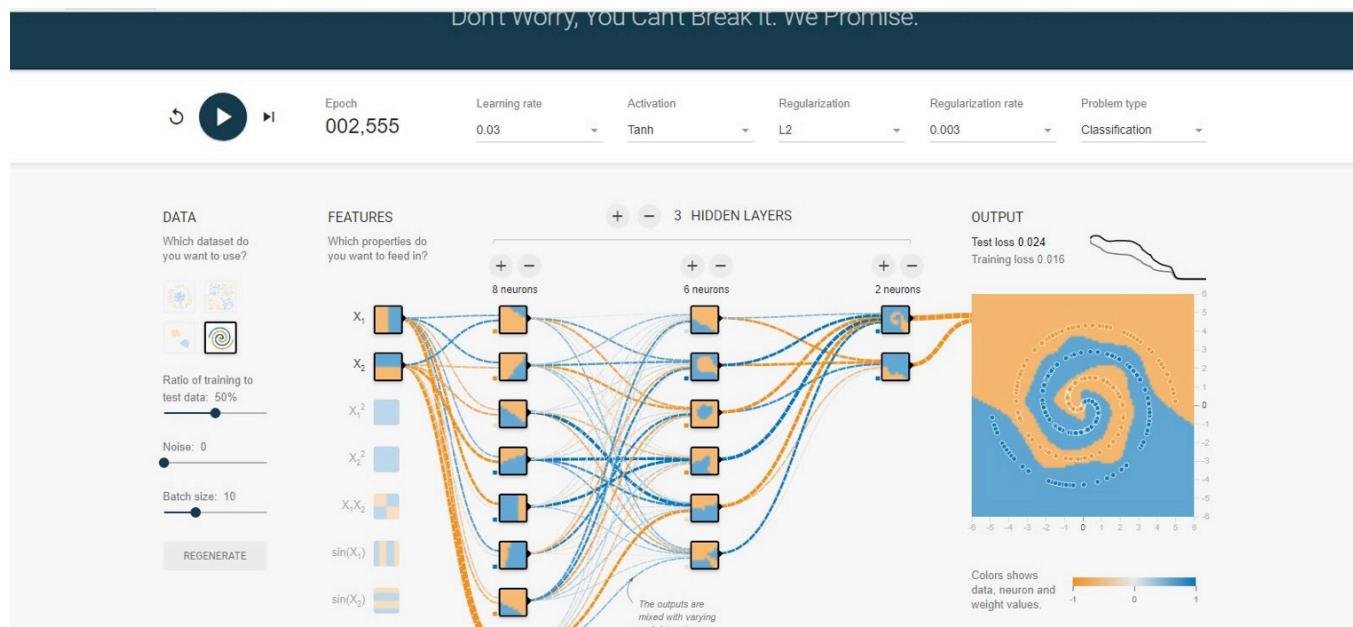
$$db^{[J]} = \frac{1}{m} \sum dZ^{[J]}$$

$$W_{new}^{[J]} = W^{[J]} - \alpha dW^{[J]}$$

$$b_{new}^{[J]} = b^{[J]} - \alpha db^{[J]}$$

# MLP Visualization

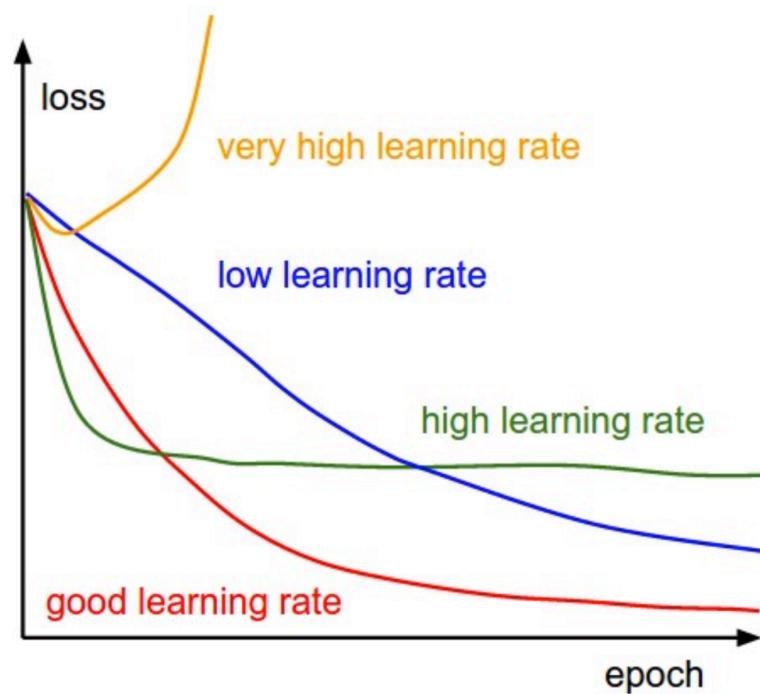
<https://playground.tensorflow.org>



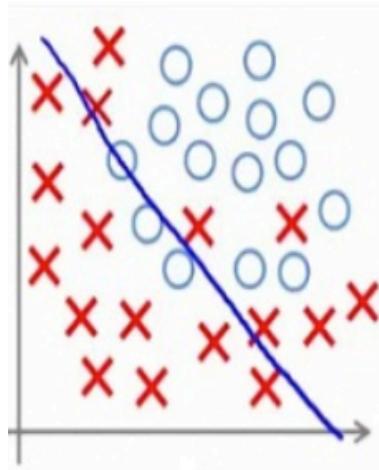
# MLP Hyperparameters

- Learning rate
- Number of iterations
- Number of Layers
- Number of units per layer
- Activation functions

# Learning Rate

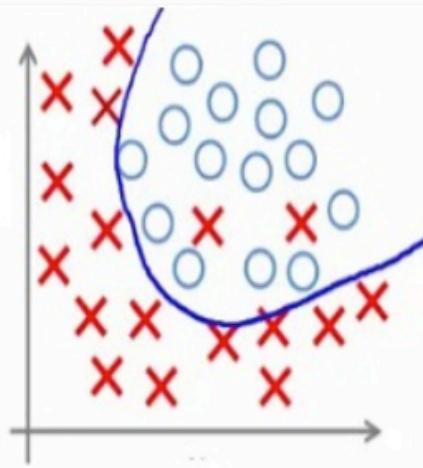


# Overfitting

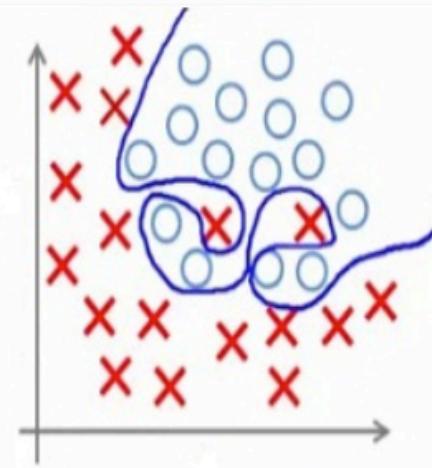


Under-fitting

(too simple to  
explain the  
variance)



Appropriate-fitting



Over-fitting

(forcefitting – too  
good to be true)

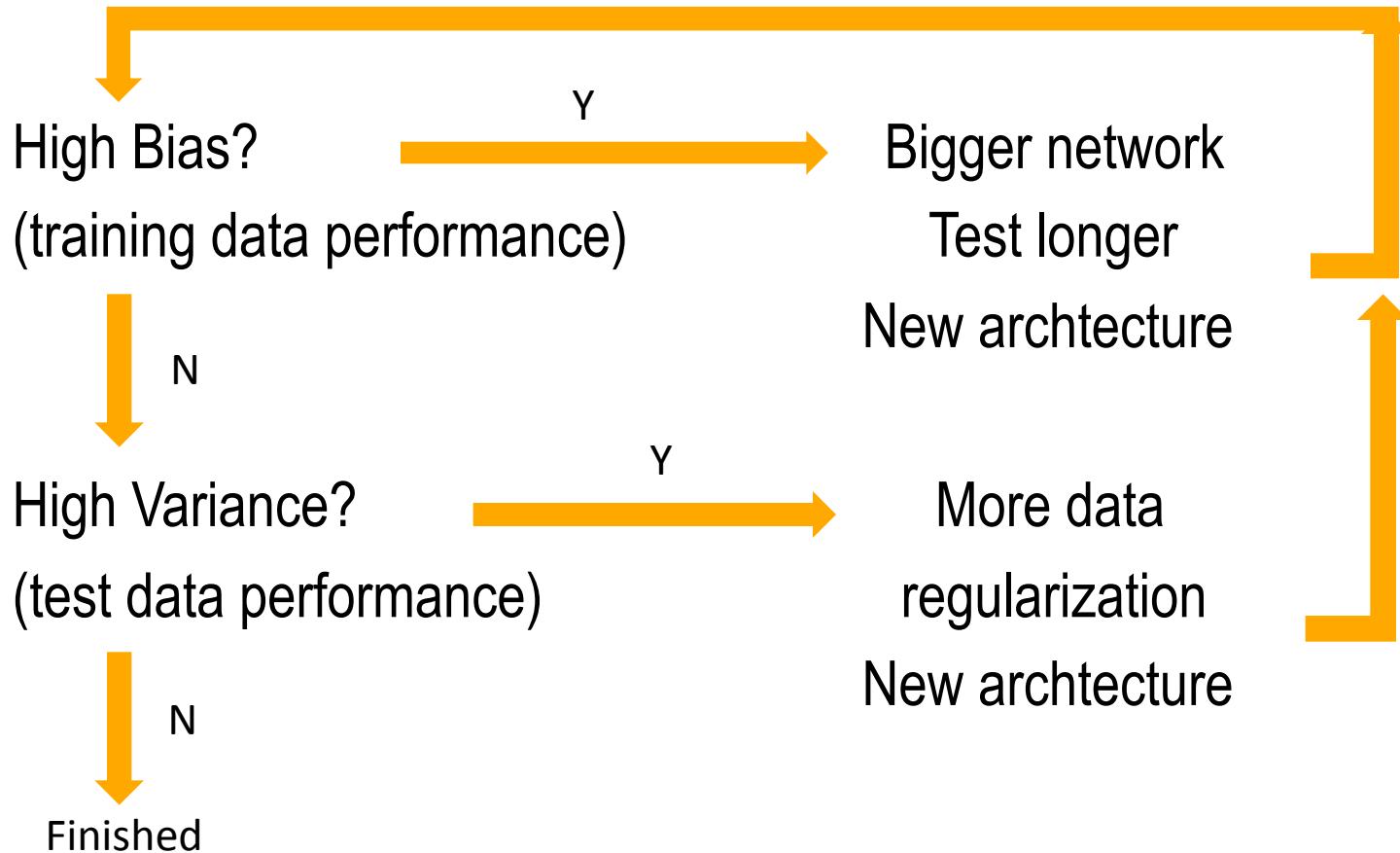
# Bias and Variance

	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none"> <li>• High training error</li> <li>• Training error close to test error</li> <li>• High bias</li> </ul>	<ul style="list-style-type: none"> <li>• Training error slightly lower than test error</li> </ul>	<ul style="list-style-type: none"> <li>• Very low training error</li> <li>• Training error much lower than test error</li> <li>• High variance</li> </ul>
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	<ul style="list-style-type: none"> <li>• Complexify model</li> <li>• Add more features</li> <li>• Train longer</li> </ul>		<ul style="list-style-type: none"> <li>• Perform regularization</li> <li>• Get more data</li> </ul>

# Bias and Variance

- If your model presents high bias:
  - Try a larger NN (> hidden layers, > units)
  - Try a different architecture that better fits your problem
  - Run longer
  - Try different optimization algorithms
- If your model presents high variance:
  - Get more data
  - Try regularization
  - Try a different architecture that better fits your problem

# Guidelines for a ML algorithm



# Techniques to reduce overfitting

- Regularization
- Data augmentation
- Early stopping (stop iterating if the dev error starts to increase).  
May result in underfitting.

# L2 Regularization

- Include a term in the cost function in order to keep NN weights small. Some will be close to 0, reducing the number of active hidden units (simpler networks)
- W influences the value of z. The activation function on small z has a more linear behavior (less overfitting)

$$J(W, b) = -\frac{1}{m} \sum L(y, \hat{y}) + \frac{\lambda}{2m} \sum_l \sum_i \sum_j (w_{ji}^{[l]})^2$$

$$dw^{[l]} = (from\_bp) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$$

# Dropout Regularization

- Inverted dropout: Randomly block a fraction  $p$  of units from each hidden layer (make  $a=0$ ), at each iteration of the training process (no dropout to test!)
- Normalize the remaining by dividing  $a/(1-p)$
- As any unit can be blocked, the network tends to get more balanced weights
- Larger layers may have larger dropout fractions
- Input layer usually has no dropout

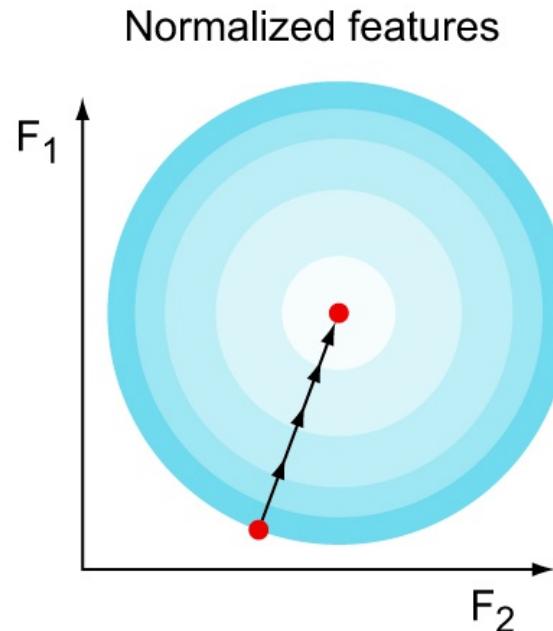
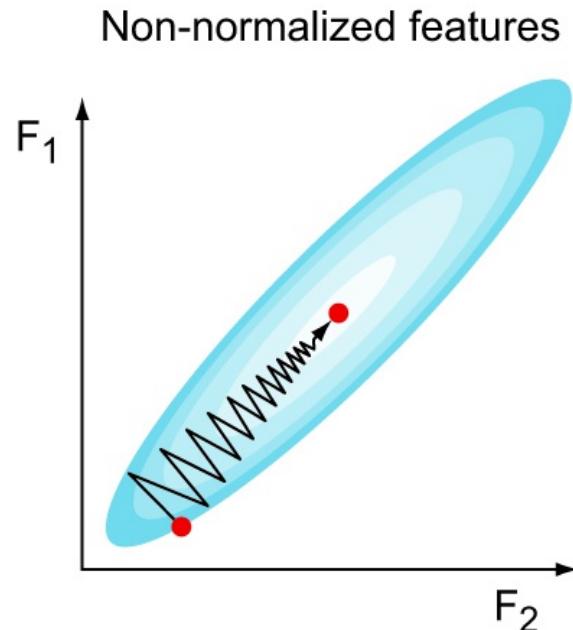
# Techniques to speedup convergence

- Normalization
- Momentum
- Mini-batch descent
- Initialization

# Normalization

- Bring input variables into same scale:  $(x - \text{mean})/\text{stdev.}$
- Training may speed up.

Gradient descent with and without feature scaling



# Momentum

- Use momentum to smooth the gradients and speed up convergence. Momentum considers the history of gradients

$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$

$$v_{db} = \beta v_{db} + (1 - \beta) db$$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

- Other methods to update parameters: RMSprop, Adam, Adaptative learning rate...

# RMSProp

- RMSprop adapts the step size individually for each weight over time, so that we accelerate learning in the direction that we need

- Parameter vector:

$$\theta^{(t)} = (b^{(t)}, w^{(t)})$$

- Gradient:

$$\nabla \mathcal{L}(\theta^{(t)}) = \left[ \frac{\partial \mathcal{L}}{\partial b}, \frac{\partial \mathcal{L}}{\partial w} \right]$$

- Variability in each coordinate:

$$S_b^{(t+1)} = \gamma S_b^{(t)} + (1 - \gamma) \left[ \frac{\partial \mathcal{L}}{\partial b} \right]^2, S_w^{(t+1)} = \gamma S_w^{(t)} + (1 - \gamma) \left[ \frac{\partial \mathcal{L}}{\partial w} \right]^2$$

- Updating:

$$w^{(t+1)} = w^{(t)} - \alpha \frac{1}{\sqrt{S_w^{(t+1)}} + \varepsilon} \frac{\partial \mathcal{L}}{\partial w} \quad b^{(t+1)} = b^{(t)} - \alpha \frac{1}{\sqrt{S_b^{(t+1)}} + \varepsilon} \frac{\partial \mathcal{L}}{\partial b}$$
$$\varepsilon = 10^{-6}$$

# Adam

ADAM = Adaptive Moment Estimation

- Combine momentum and RMSProp
- Define 2 exponentially smoothed series:
  - Of gradient, using momentum:

$$\mathbf{V}^{(t)} = (1 - \beta) \nabla \mathcal{L}(\mathbf{w}^{(t)}) + \beta \mathbf{V}^{(t-1)}$$

- Of the squared gradient, like RMSProp:

$$S_w^{(t+1)} = \gamma S_w^{(t)} + (1 - \gamma) \left[ \frac{\partial \mathcal{L}}{\partial w} \right]^2$$

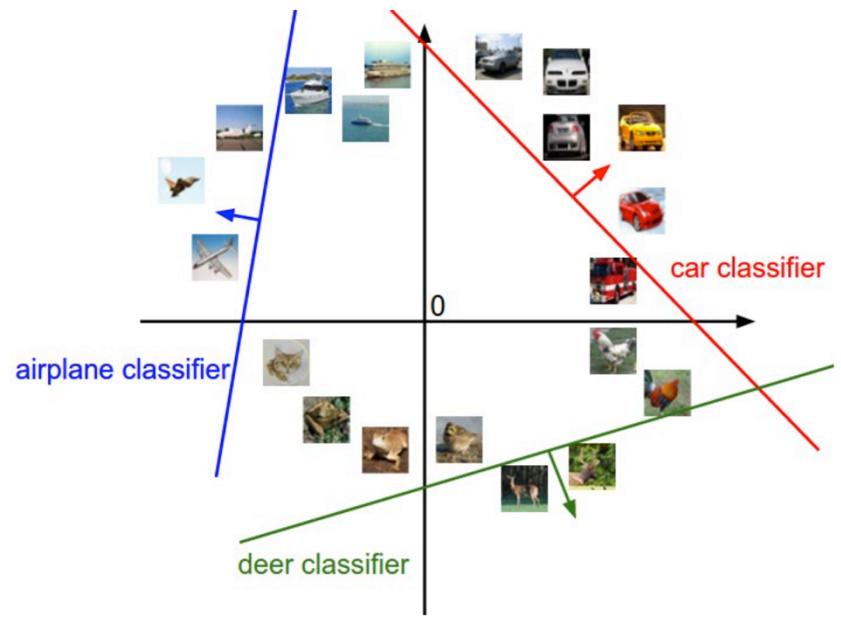
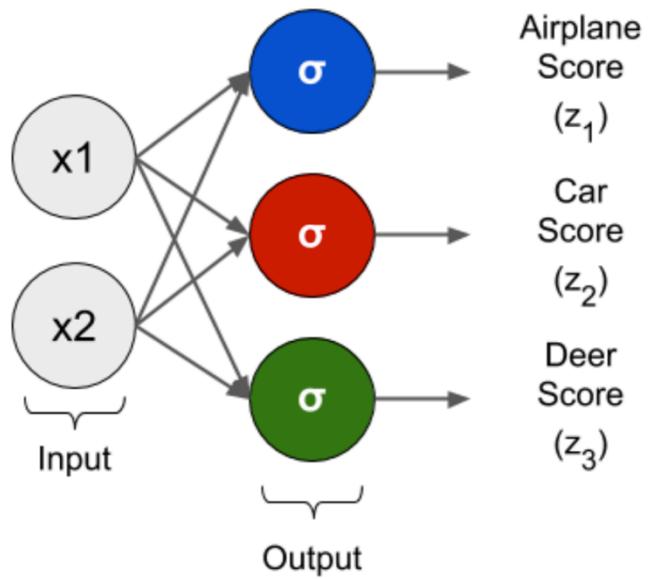
- Update:

$$w^{(t+1)} = w^{(t)} - \frac{\alpha}{\sqrt{S_w^{(t+1)}} + \varepsilon} \mathbf{V}^{(t)}$$

# Other useful techniques

- Vanishing/exploding gradients may occur in deep NN, slowing down training.  
Initializing the weights properly may avoid some of these problems:
  - $W^{[l]} = \text{random} * \text{sqrt}(2/n^{[l-1]})$  for ReLU
  - $W^{[l]} = \text{random} * \text{sqrt}(1/n^{[l-1]})$  for tanh
- Use mini-batch gradient descent+batch normalization: partition training set into small batches and do many forward-backward loops for each epoch
- Batch normalization standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.
- Size of batches is a hyperparameter: size=m is the batch version; size=1 is the stochastic gradient descent. Size must fit in memory

# Multiple classes



# Multiple classes

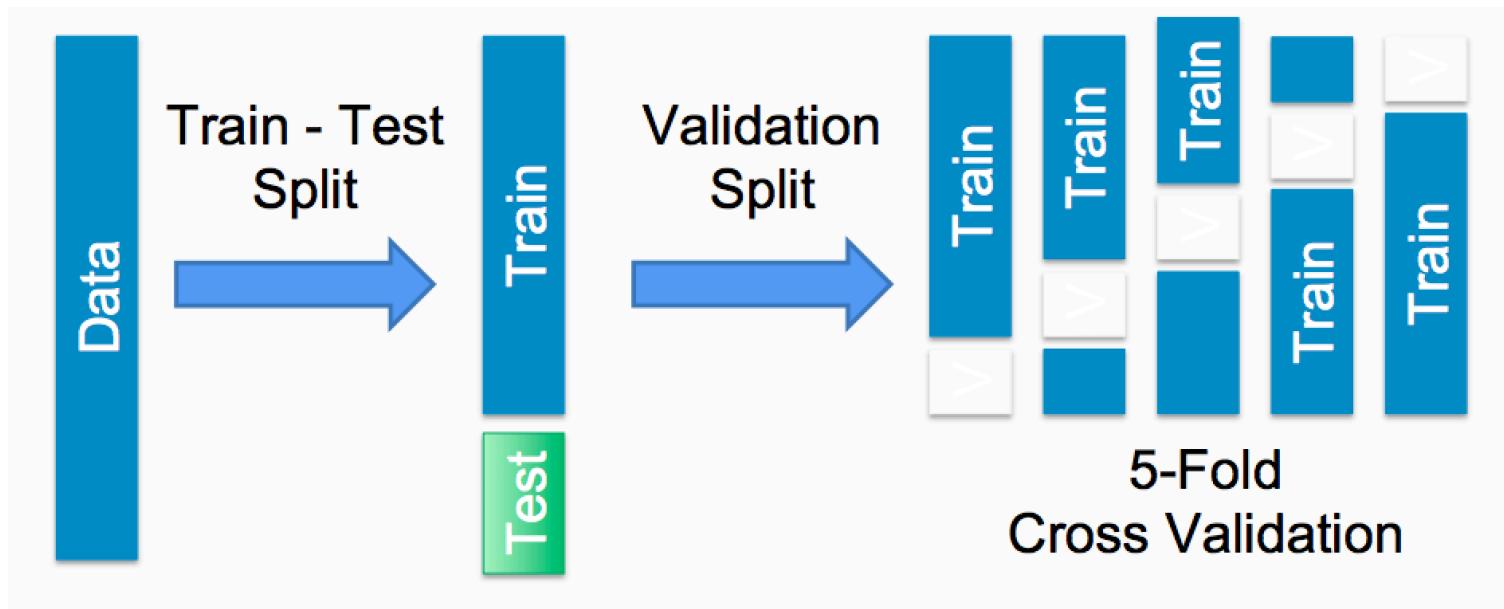
- In order to classify into n classe, use a n-unit output layer
- Each unit's output should reflect the probability of the input being each class
- Use the softmax activation function:

$$\alpha_i = \hat{y}_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

and the Loss Function (Maximum Likelihood Estimator):

$$L(y, \hat{y}) = -\sum_{j=1}^n y_j \log \hat{y}_j; dz = \hat{y} - y$$

# Data Split Strategies



# References and acknowledgements

Some of these slides were inspired or adapted from courses and presentations given by Andrew Ng, Camila Laranjeira, Fei-Fei Li, Flávio Figueiredo, Hugo Oliveira, Jefersson dos Santos, Justin Johnson, Keiller Nogueira, Pedro Olmo, Renato Assunção, Serena Yeung.

Reference courses include *Machine Learning* and *Deep Learning* CS230 and CS231 from Stanford University, *Deep Learning* and *Hands-on Deep Learning* from UFMG, *Deep Learning* CS498 from Un. Of Illinois.