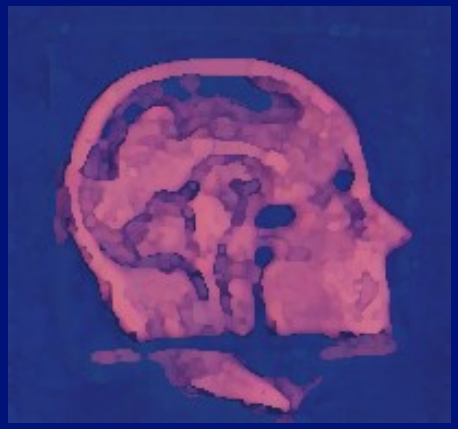
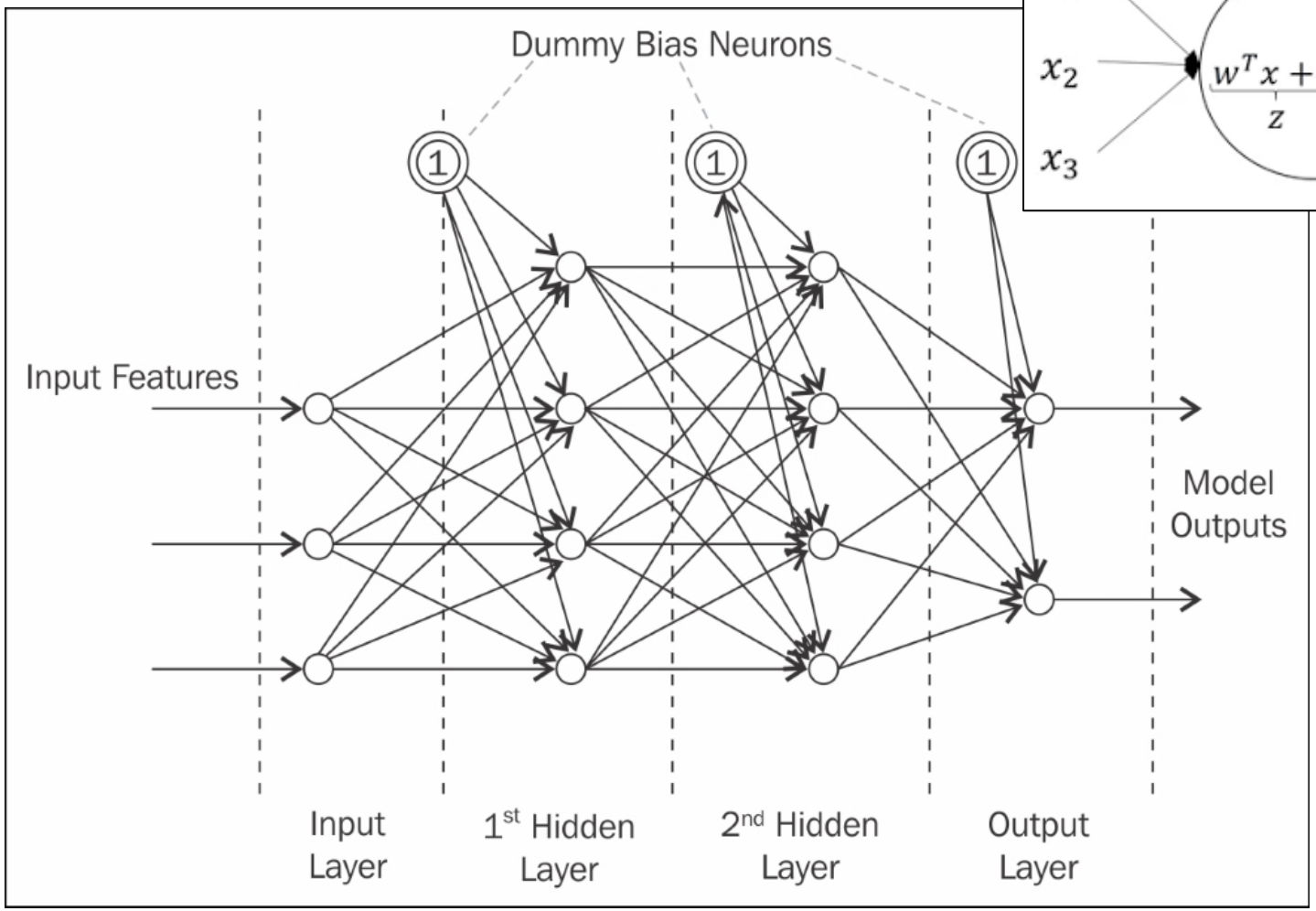
Deep Learning in Computer Vision

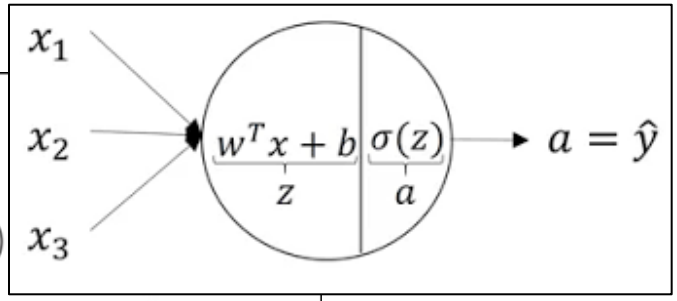


Alexei Manso Corrêa Machado
Pontifical Catholic University of Minas Gerais – D. Computer Science

2/2020

The Multi-Layer Perceptron

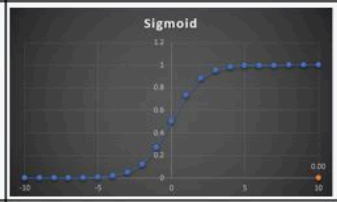
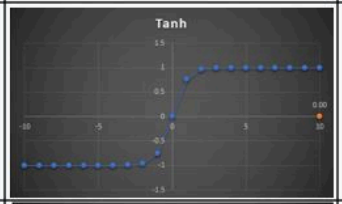
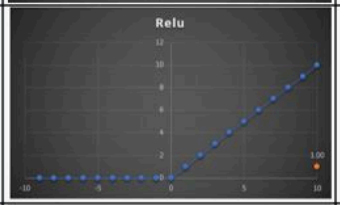




Notation

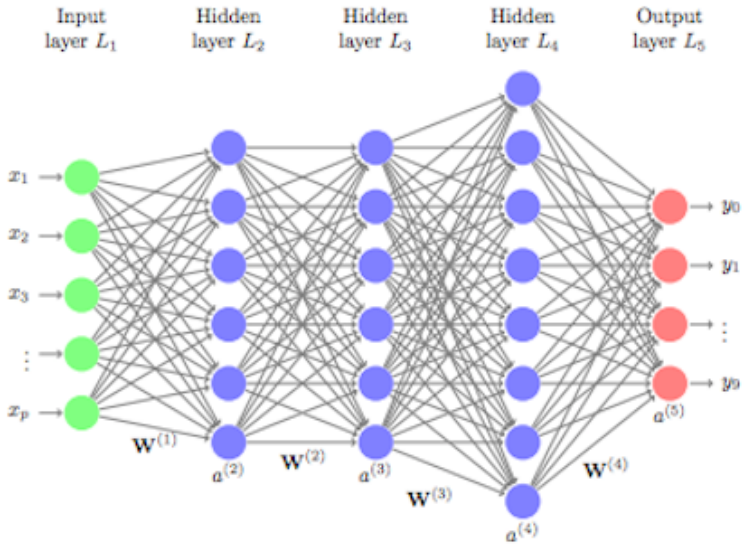
- X is the $n^{[0]}$,m data matrix of m samples
- Y is the $n^{[L]}$,m label matrix of m samples
- [l] is the layer index, l=0…L
- L is the number of layers
- $x^{(i)}$ denotes the i-th sample
- $x^{(i)}_j$ denotes the j-th variable of the i-th sample
- Each layer l has $n^{[l]}$ units
- $W^{[l]}$ is 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 $g(Z^{[l]})$
- $X=A^{[0]}$; $Y\hat{}=A^{[L]}$

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

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

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

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

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

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

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

MLP Gradient Descent – Backward/ Update

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

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

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

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

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

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

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

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

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

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

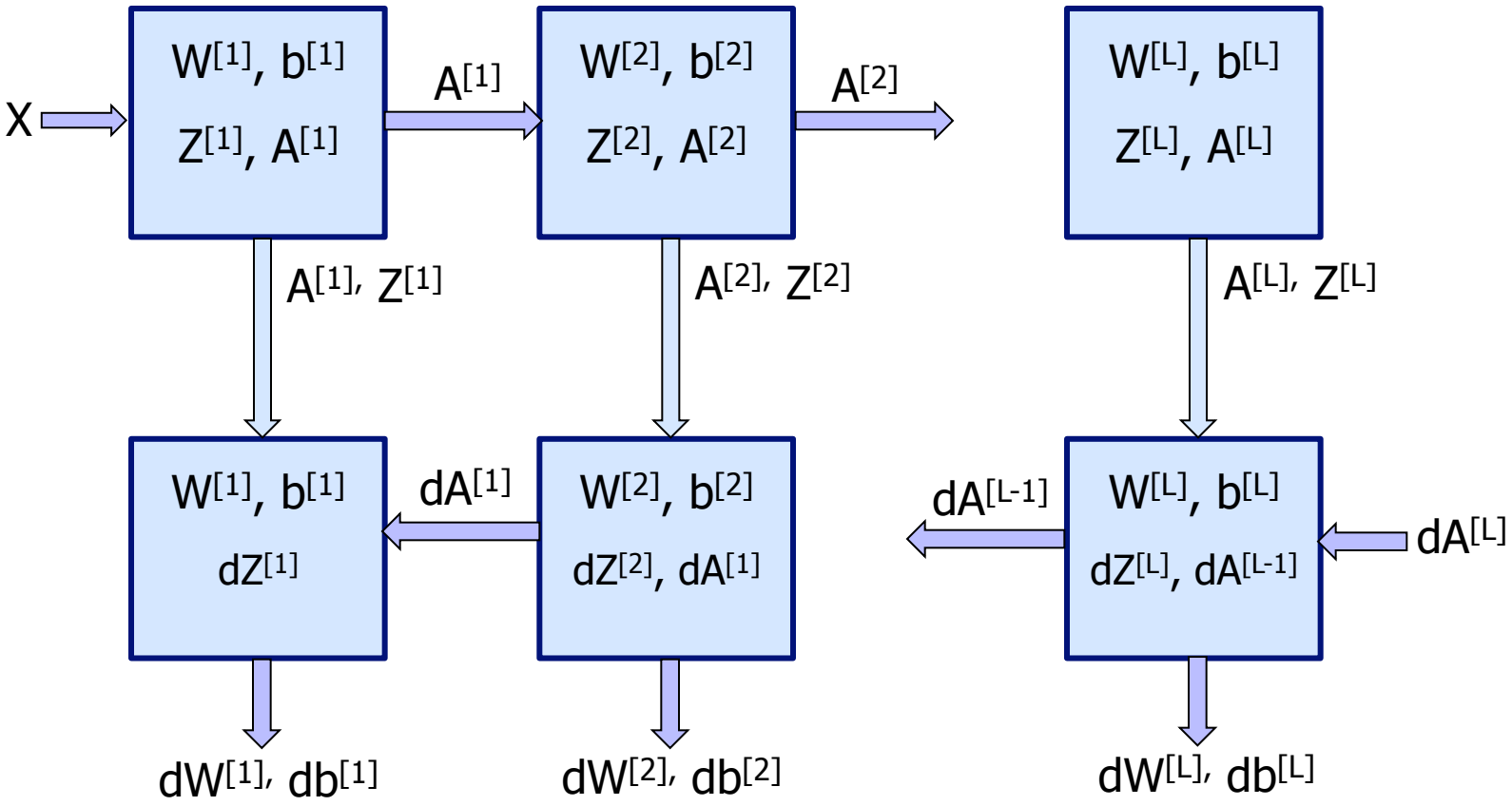
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

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

$$W^{[l]}_{new} = W^{[l]} - \alpha dW^{[l]}$$

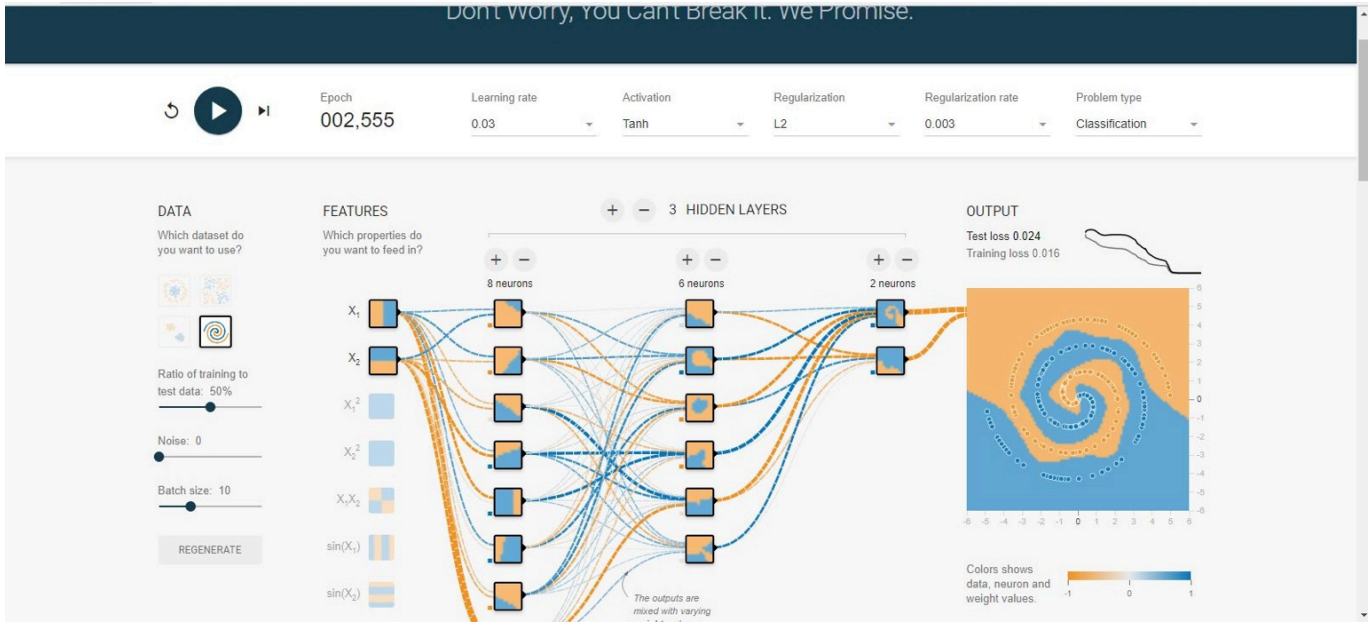
$$b^{[l]}_{new} = b^{[l]} - \alpha db^{[l]}$$

Using caches



MLP Visualization

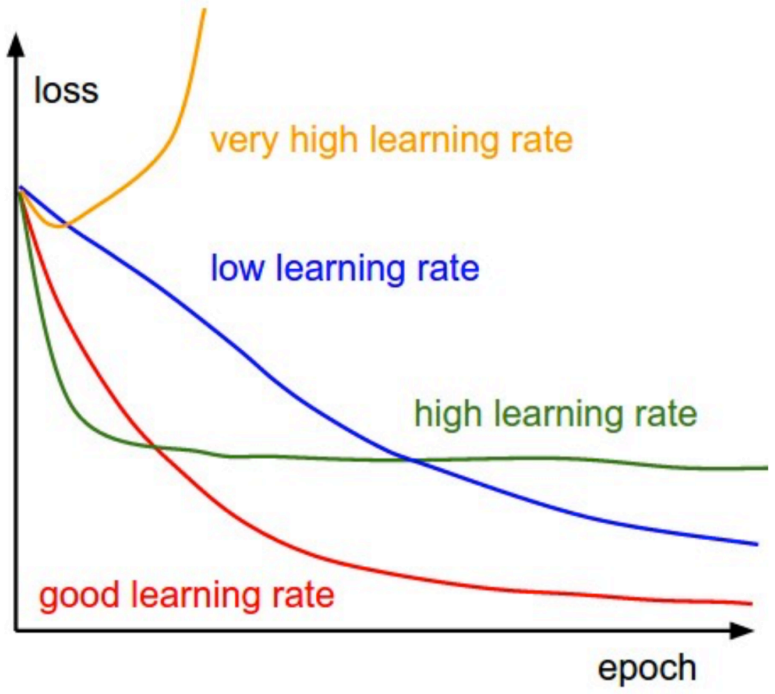
https://playground.tensorflow.org



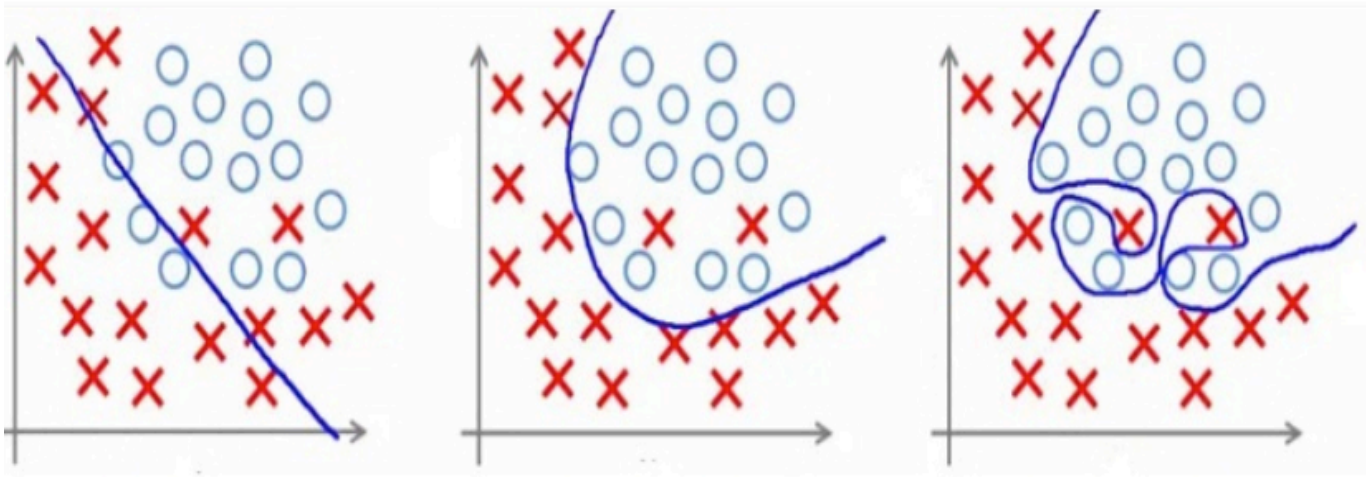
MLP Hyperparameters

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

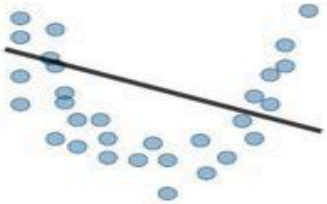
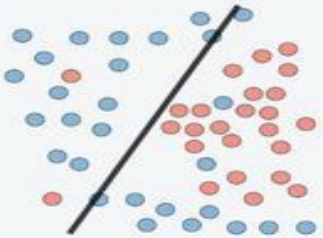
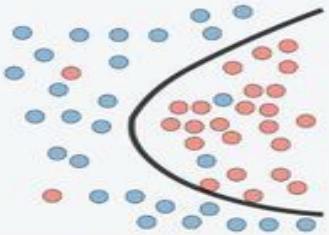
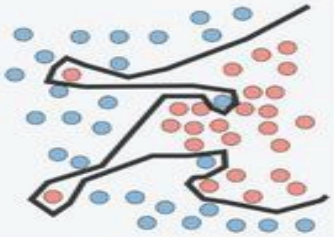
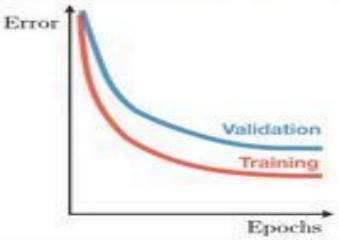
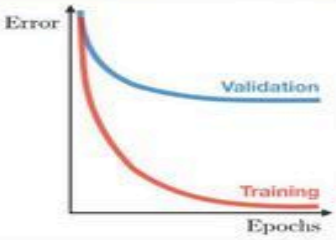
Learning Rate



Overfiting


Under-fitting
(too simple to explain the variance)
Appropriate-fitting
Over-fitting
(forcefitting -- too good to be true)

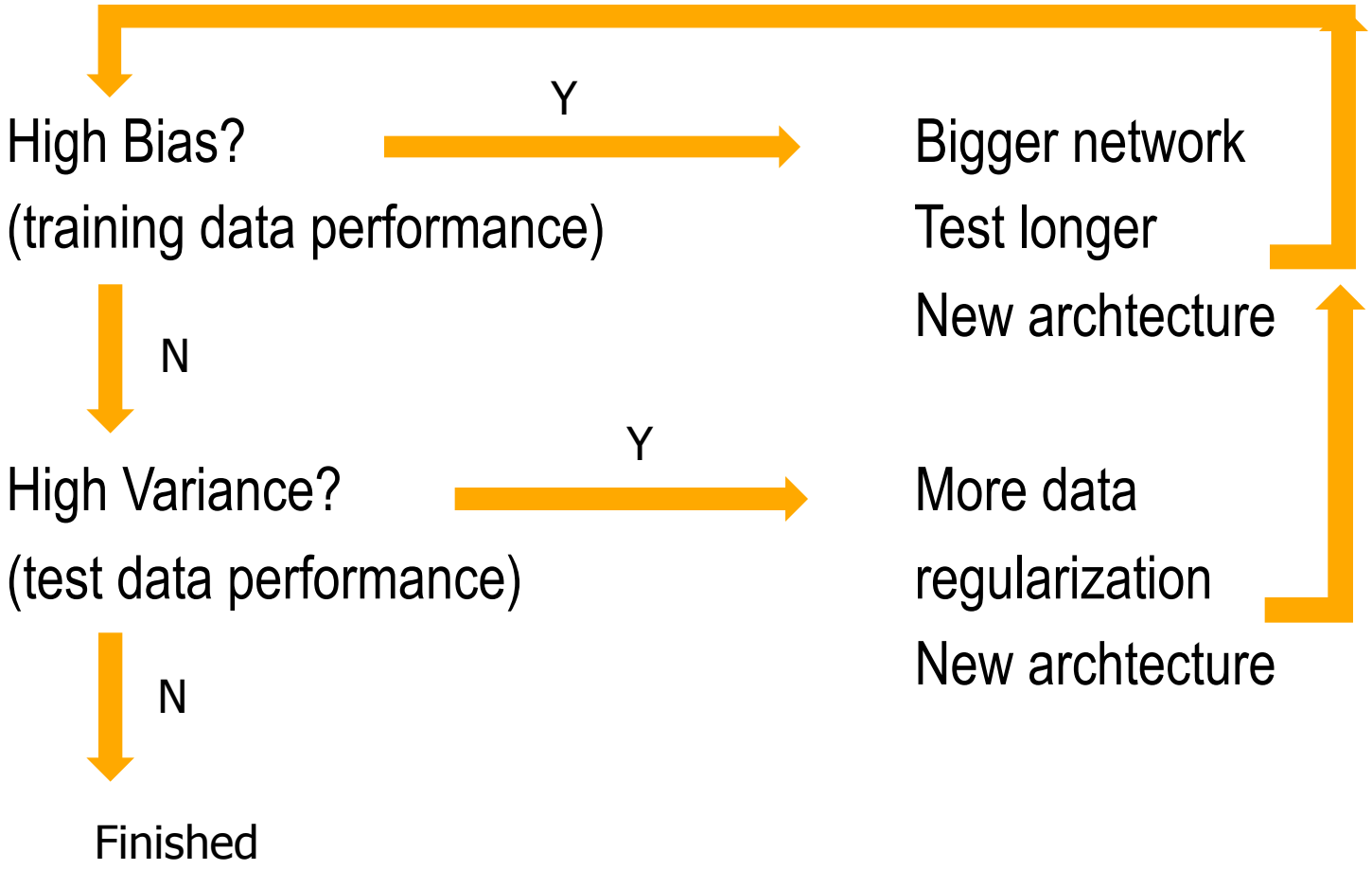
Bias and Variance

	Underfitting	Just right	Overfitting
Symptoms	• High training error • Training error close to test error • High bias	• Training error slightly lower than test error	• Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	• Complexify model • Add more features • Train longer		• Perform regularization • Get more data

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 of small z has a more linear behavior (less overfitting)

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

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

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

Dropout Regularization

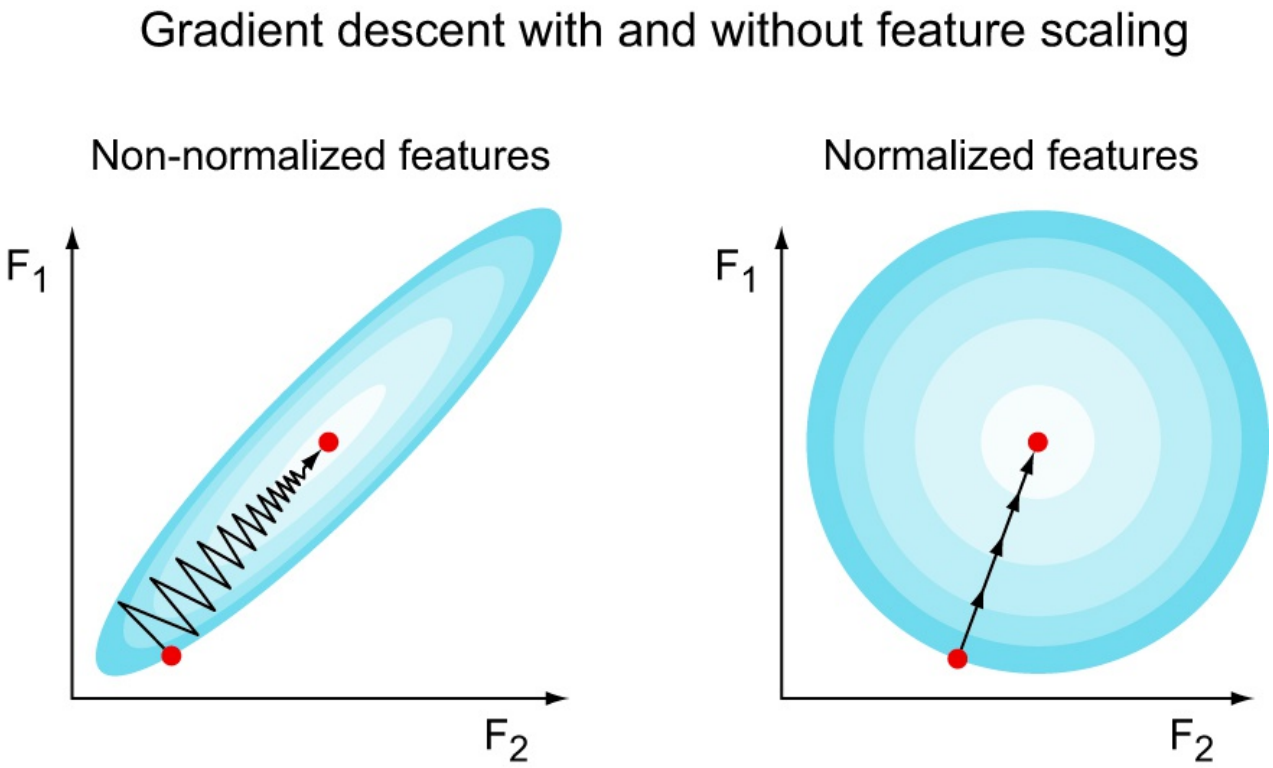
- Inverted dropout: Randomly block a fraction *p* of units from each hidden layer (make a=0), at each iteration of the training process (no dropout to test!)
- Normalize the remaining by dividing *a/(1-p)*
- As any unit can be blocked, the network tends to get more balanced weights
- Larger layers may 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-mean)/stdev.
- Training mayspeed up.



Momentum

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

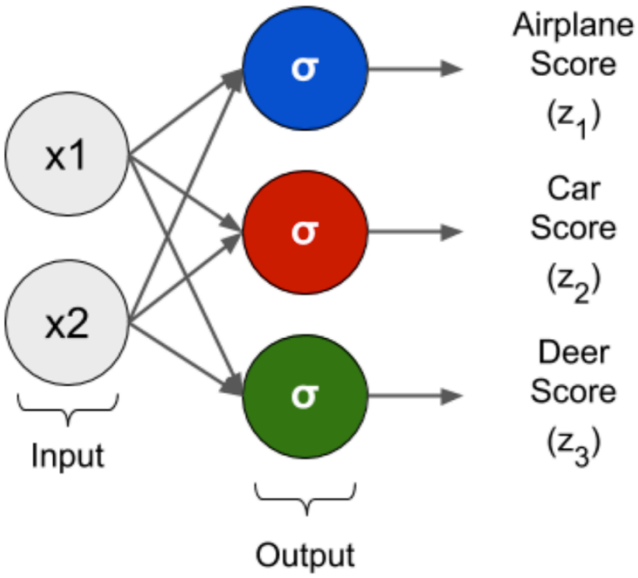
$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$
$$v_{db} = \beta v_{db} + (1 - \beta) db$$
$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

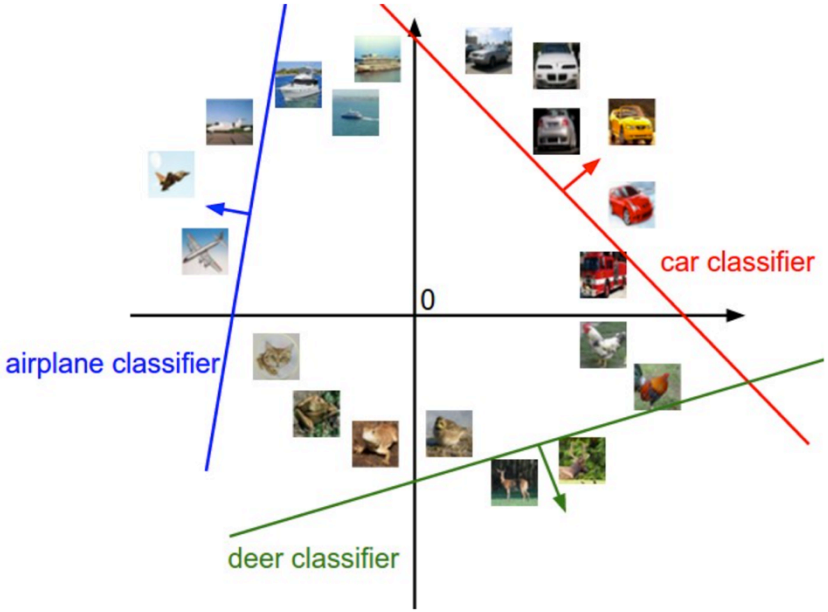
- Other methods to update parameters: RMSprop, Adam, Adaptative learning rate...

Other useful techniques

- Vanishing/exploding gradients may occur in deep NN, slowing down training. Initializing the weights properly may avoid some of these problems:
  - $W^{[l]}$=random*sqrt(2/$n^{[l-1]}$) for ReLU
  - $W^{[l]}$=random*sqrt(1/$n^{[l-1]}$) for tanh
- Use mini-batch gradient descent: partition training set into small batches and do many forward-backward loops for each epoch
- 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

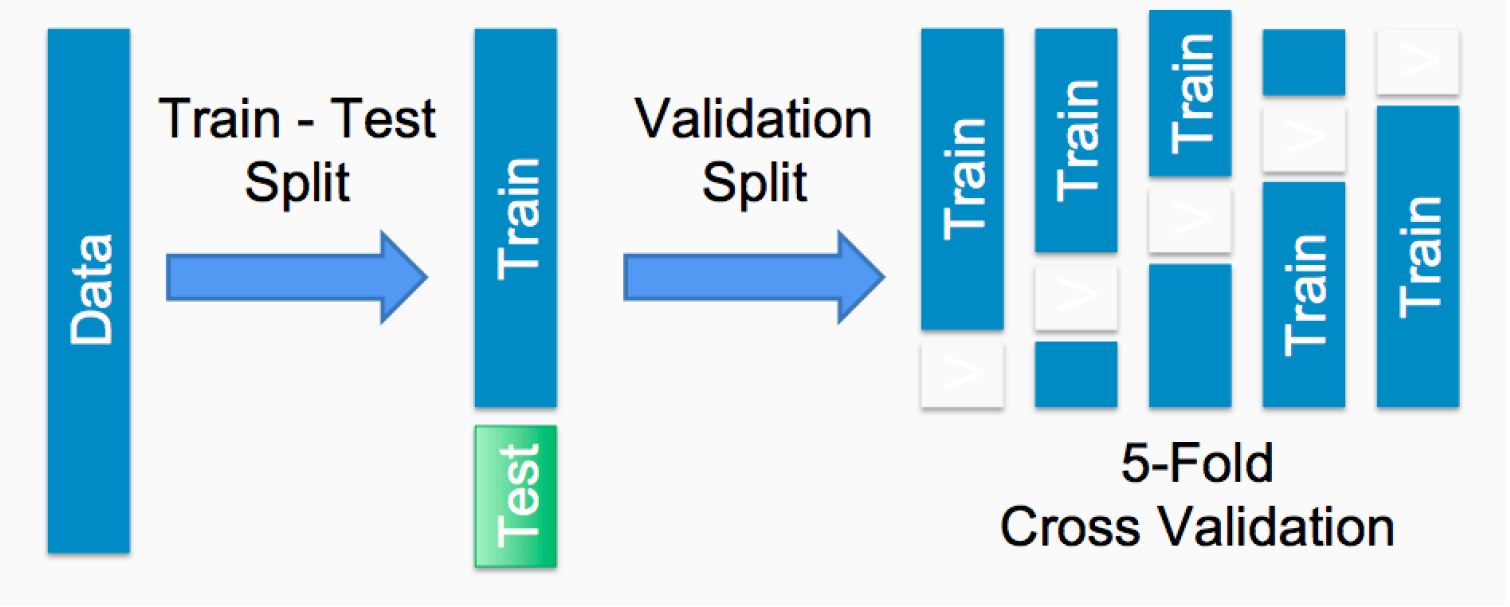
- In order to classify into n classe, use a n-unit output layer
- Each unit's output should reflect the probability of the input being each class
- Use the softmax activation function:

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

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, Fei-Fei Li, Flávio Figueiredo, Jefersson dos Santos, Justin Johnson, Pedro Olmo, Renato Assunção, Serena Yeung.

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