Hyperparameter tuning, Regularization and Optimization

A quick overview of Coursera's 2nd specialization course

George Zoto
gezotos@gmail.com

The 2nd course of the Deep Learning Specialization

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

deeplearning.ai

About this Course

This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow.

After 3 weeks, you will:

- Understand industry best-practices for building deep learning applications.
- Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking,
- Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.
- Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance
- Be able to implement a neural network in TensorFlow.

This is the second course of the Deep Learning Specialization.

Setup

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. You can write and execute code, save and share your analyses, and access powerful computing resources, all for free from your browser.

https://colab.research.google.com

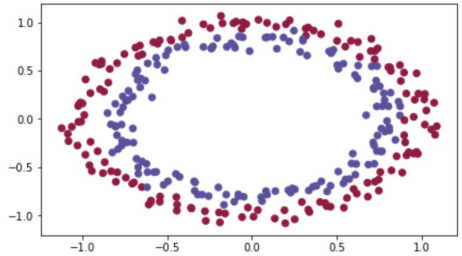
Week 1: Practical aspects of Deep Learning

Setting up your Machine Learning Application

Regularizing your neural network

Setting up your optimization problem

Setting up your ML Application: Weight Initialization

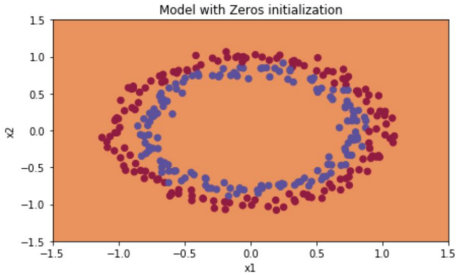


You would like a classifier to separate the blue dots from the red dots.

1 - Neural Network model

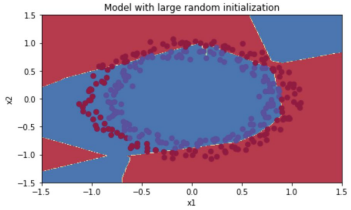
You will use a 3-layer neural network (already implemented for you). Here are the initialization methods you will experiment with:

- *Zeros initialization* -- setting `initialization = "zeros"` in the input argument.
- *Random initialization* -- setting `initialization = "random"` in the input argument. This initializes the weights to large random values.
- *He initialization* -- setting `initialization = "he"` in the input argument. This initializes the weights to random values scaled according to a paper by He et al., 2015.



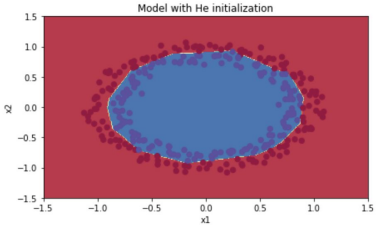
The model is predicting 0 for every example.

In general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and you might as well be training a neural network with $n^{[l]} = 1$ for every layer, and the network is no more powerful than a linear classifier such as logistic regression.



Observations:

- The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example. Indeed, when $\log(a^{[3]}) = \log(0)$, the loss goes to infinity.
- Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.
- If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.



Observations:

- The model with He initialization separates the blue and the red dots very well in a small number of iterations.

Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

What you should remember from this notebook:

- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Setting up your ML Application: Regularization

Problem Statement: You have just been hired as an AI expert by the French Football Corporation. They would like you to recommend positions where France's goal keeper should kick the ball so that the French team's players can then hit it with their head.

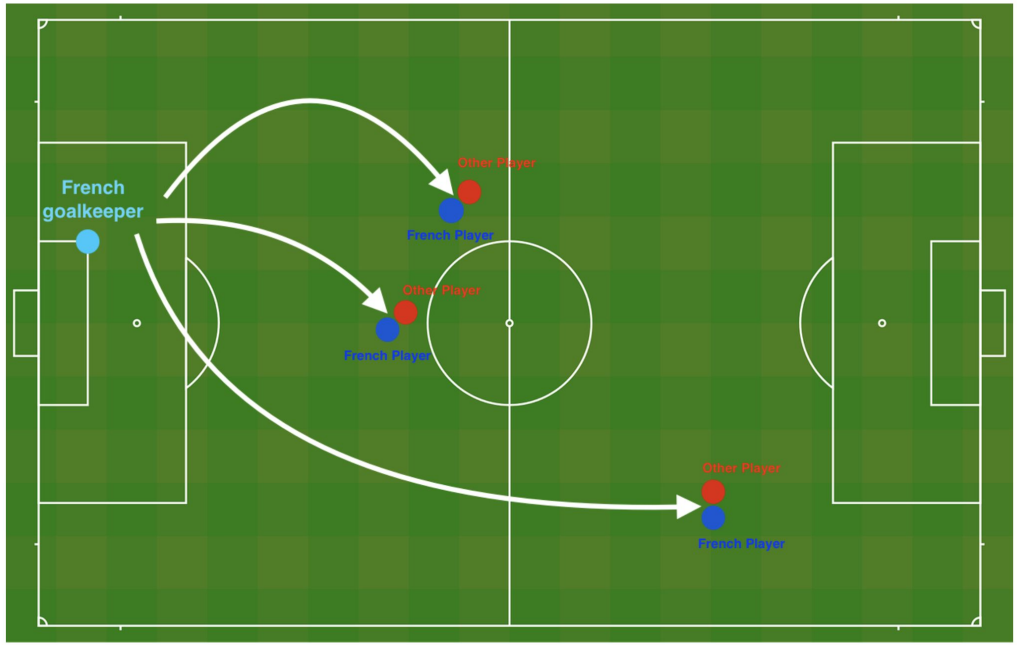
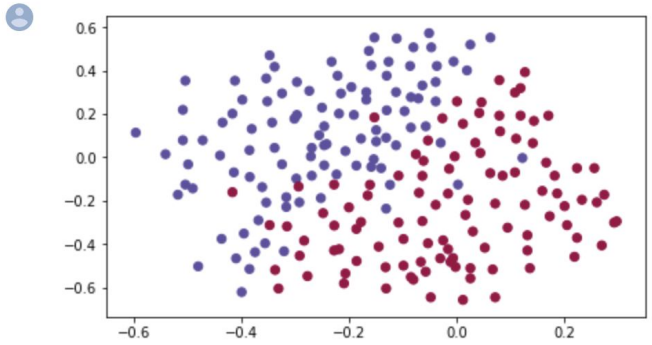


Figure 1: **Football field**
The goal keeper kicks the ball in the air, the players of each team are fighting to hit the ball with their head

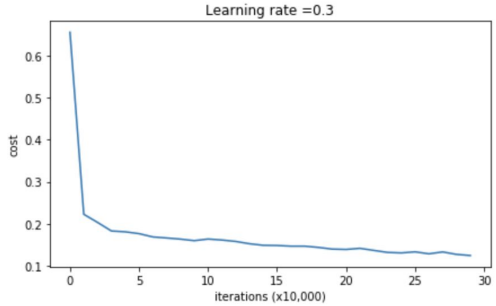


Each dot corresponds to a position on the football field where a football player has hit the ball with his/her head after the French goal keeper has shot the ball from the left side of the football field.

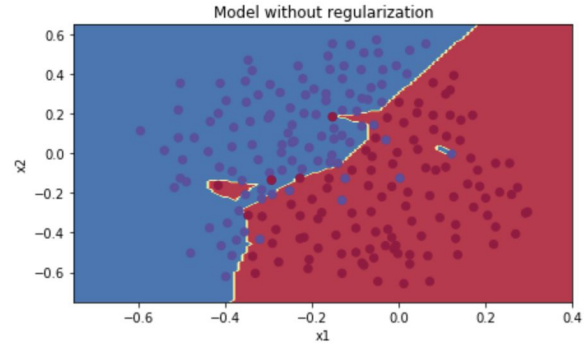
- If the dot is blue, it means the French player managed to hit the ball with his/her head
- If the dot is red, it means the other team's player hit the ball with their head

Your goal: Use a deep learning model to find the positions on the field where the goalkeeper should kick the ball.

Setting up your ML Application: Regularization

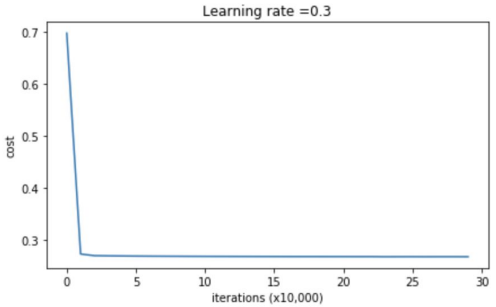



```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

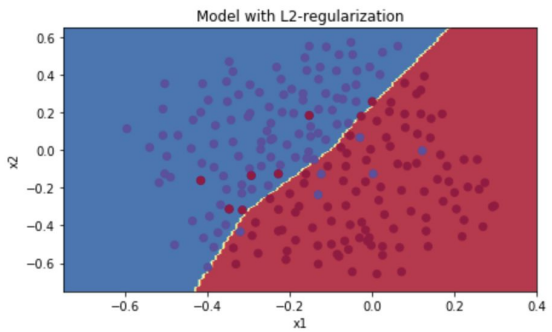


$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

The non-regularized model is obviously overfitting the training set. It is fitting the noisy points! Lets now look at two techniques to reduce overfitting.




```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

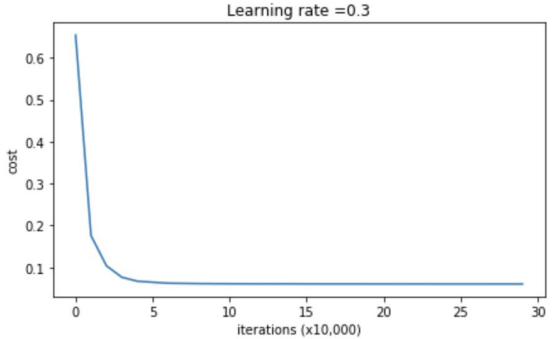


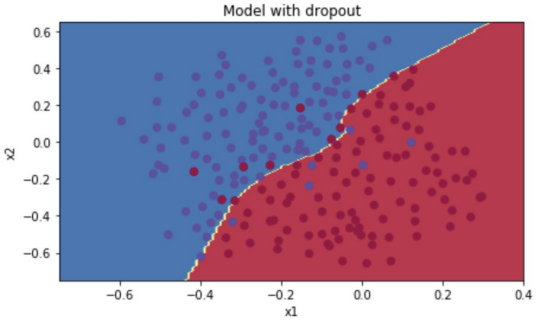
$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

Observations:

- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

Setting up your ML Application: Regularization






```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```

Note:

- A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.
- Deep learning frameworks like tensorflow, PaddlePaddle, keras or caffe come with a dropout layer implementation. Don't stress - you will soon learn some of these frameworks.

Model	Train accuracy	Test Accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

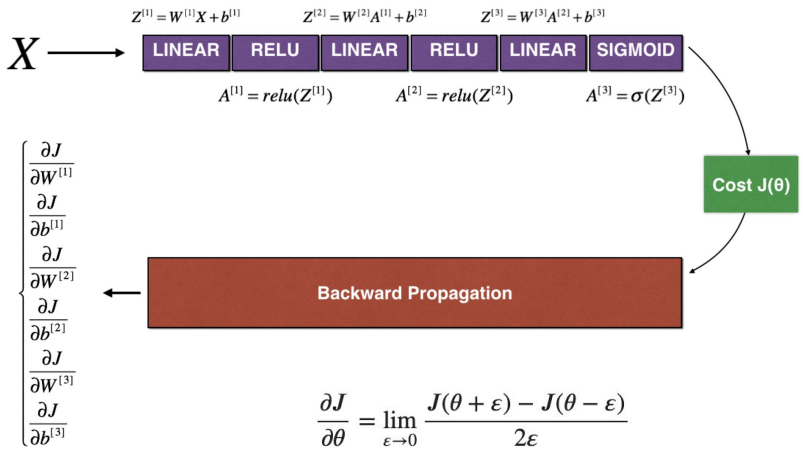
What we want you to remember from this notebook:

- Regularization will help you reduce overfitting.
- Regularization will drive your weights to lower values.
- L2 regularization and Dropout are two very effective regularization techniques.

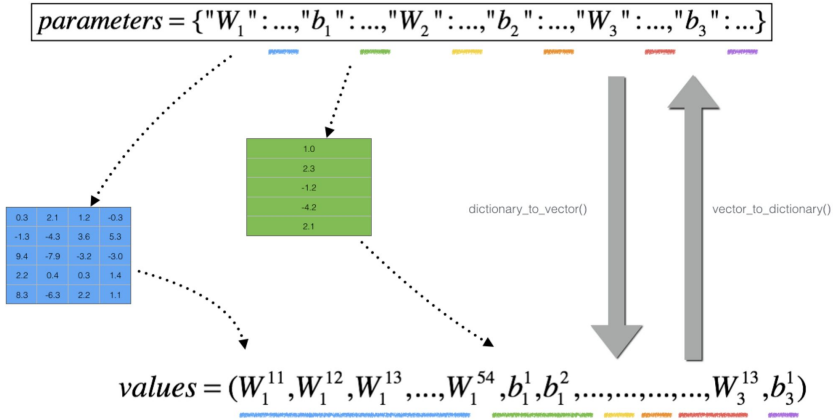
Setting up your ML Application: Gradient Checking

You are part of a team working to make mobile payments available globally, and are asked to build a deep learning model to detect fraud--whenever someone makes a payment, you want to see if the payment might be fraudulent, such as if the user's account has been taken over by a hacker.

But backpropagation is quite challenging to implement, and sometimes has bugs. Because this is a mission-critical application, your company's CEO wants to be really certain that your implementation of backpropagation is correct. Your CEO says, "Give me a proof that your backpropagation is actually working!" To give this reassurance, you are going to use "gradient checking".



$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$



For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set θ^+ to `np.copy(parameters_values)`
  2. Set θ_i^+ to $\theta_i^+ + \varepsilon$
  3. Calculate J_i^+ using to `forward_propagation_n(x, y, vector_to_dictionary(`θ^+ `))`.
- To compute `J_minus[i]`: do the same thing with θ^-
- Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

What you should remember from this notebook:

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.

$$difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2}$$

Week 2: Optimization Algorithms

Gradient Descent

Mini-Batch Gradient Descent

Momentum

Adam

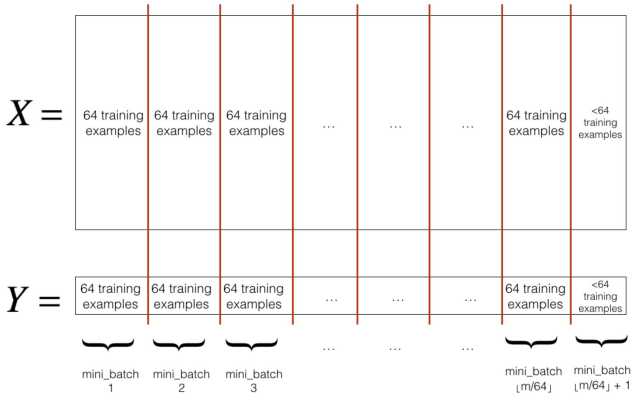
Optimization Algorithms

The gradient descent rule is, for $l = 1, \ldots, L$:

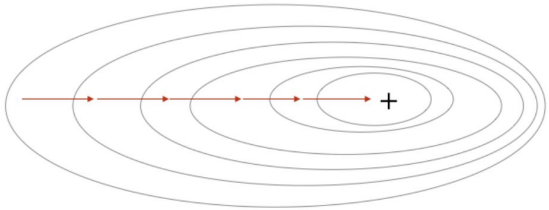
$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$

$$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \ldots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \ldots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \ldots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix} \qquad Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$

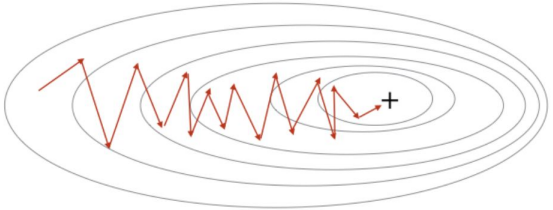
$$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \ldots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \ldots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \ldots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix} \qquad Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(m-1)} & y^{(m)} \end{pmatrix}$$



Gradient Descent



Stochastic Gradient Descent

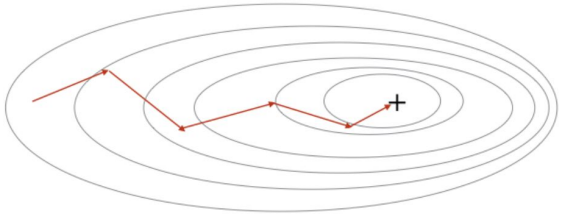


The momentum update rule is, for $l = 1, \ldots, L$:

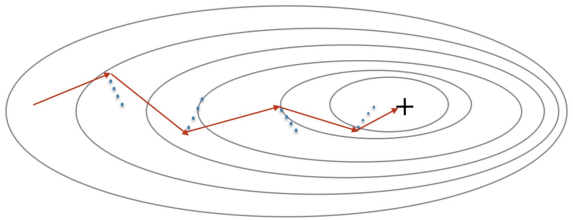
$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

Mini-Batch Gradient Descent



Momentum



Optimization Algorithms

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

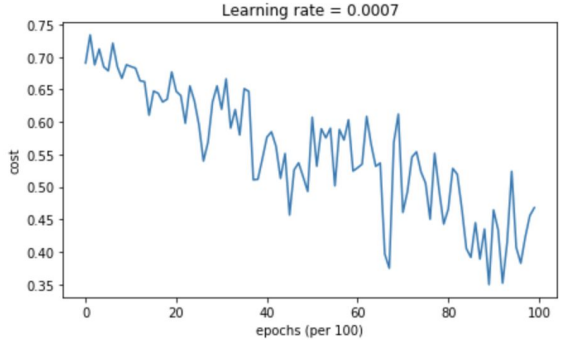
The update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s_{dW^{[l]}}^{corrected} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}} + \varepsilon} \end{cases}$$

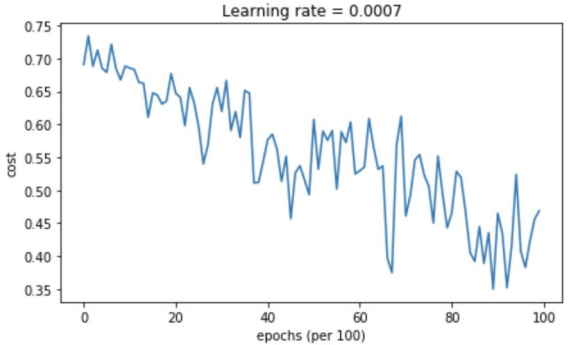
where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

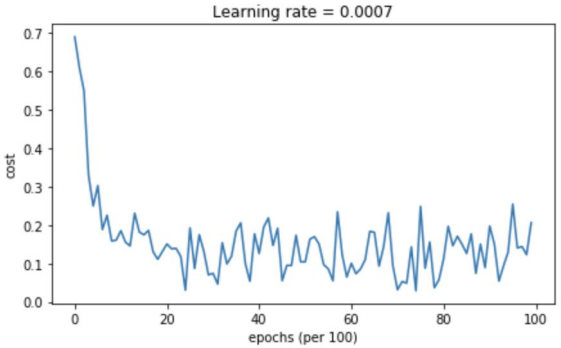
Optimization Algorithms



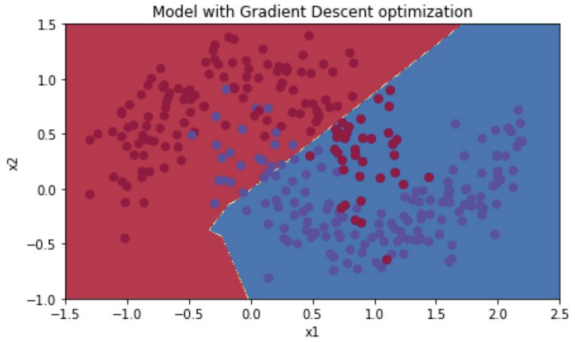
Accuracy: 0.7966666666666666

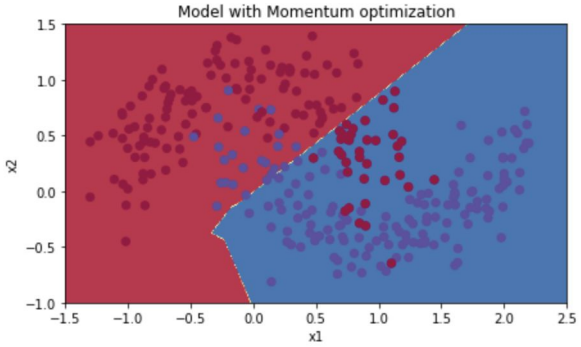


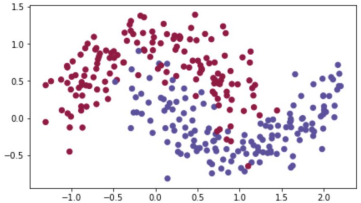
Accuracy: 0.7966666666666666



Accuracy: 0.94







Optimization method	Accuracy	Cost shape
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

Week 3: Hyperparameter Tuning, Batch Normalization and Programming Frameworks

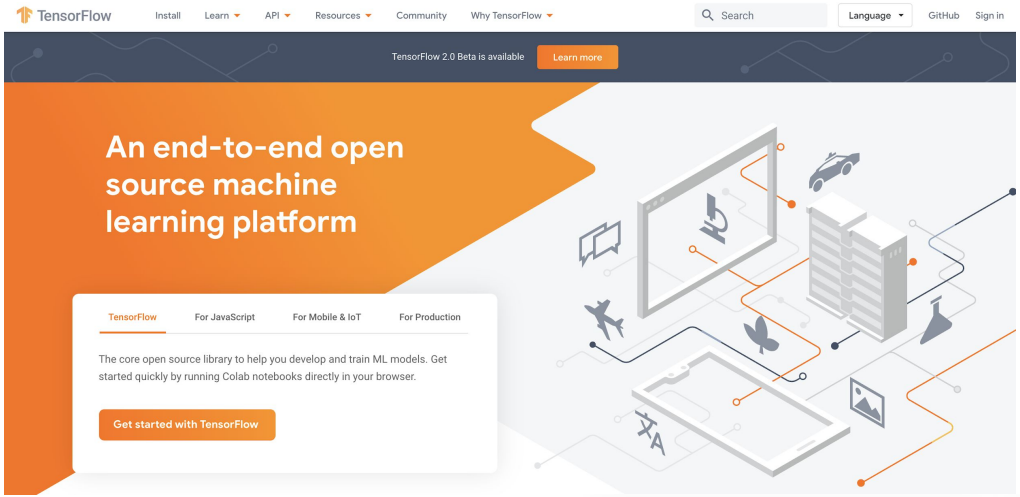
Hyperparameter Tuning

Batch Normalization

Multi-class classification

Programming Frameworks

Optimization Algorithms



https://www.tensorflow.org

For beginners

Your first neural network

Train a neural network to classify images of clothing, like sneakers and shirts, in this fast-paced overview of a complete TensorFlow program.

For experts

Generative adversarial networks

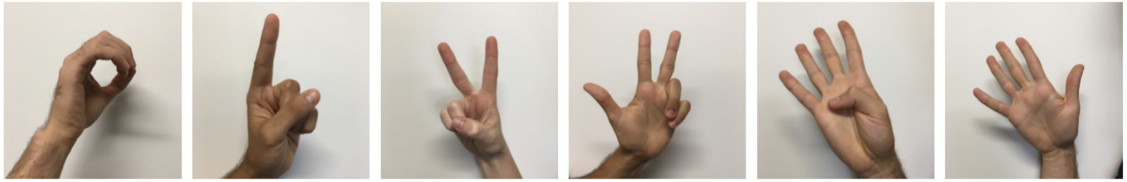
Train a generative adversarial network to generate images of handwritten digits, using the Keras Subclassing API.

For experts

Neural machine translation with attention

Train a sequence-to-sequence model for Spanish to English translation using the Keras Subclassing API.

Optimization Algorithms

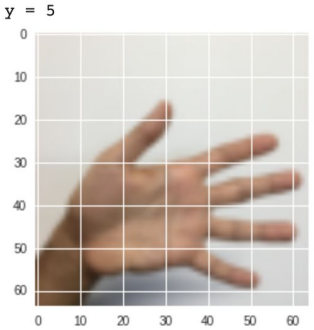


y = 0 → $\begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix}$ y = 1 → $\begin{bmatrix}0\\1\\0\\0\\0\\0\end{bmatrix}$ y = 2 → $\begin{bmatrix}0\\0\\1\\0\\0\\0\end{bmatrix}$ y = 3 → $\begin{bmatrix}0\\0\\0\\1\\0\\0\end{bmatrix}$ y = 4 → $\begin{bmatrix}0\\0\\0\\0\\1\\0\end{bmatrix}$ y = 5 → $\begin{bmatrix}0\\0\\0\\0\\0\\1\end{bmatrix}$

$y = [1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 1]$ is often converted to

$$\begin{bmatrix}0&0&0&1&0&0\\1&0&0&0&0&1\\0&1&0&0&1&0\\0&0&1&0&0&0\end{bmatrix}$$

class = 0
class = 1
class = 2
class = 3

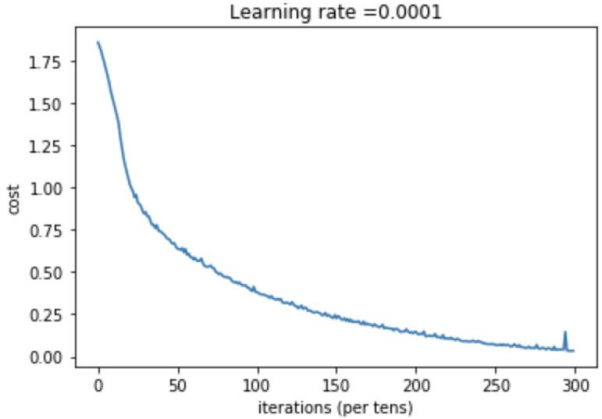


Your goal is to build an algorithm capable of recognizing a sign with high accuracy. To do so, you are going to build a tensorflow model that is almost the same one you have previously built in numpy for cat recognition (but now using a softmax output). It is a great occasion to compare your numpy implementation to t tensorflow one.

The model is *LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX*. The SIGMOID output layer has been converted to a SOFTMAX. A SOFTMAX layer generalizes SIGMOID to when there are more than two classes.

Optimization Algorithms

```
Cost after epoch 0: 1.855702
Cost after epoch 100: 1.017255
Cost after epoch 200: 0.733184
Cost after epoch 300: 0.573071
Cost after epoch 400: 0.468574
Cost after epoch 500: 0.381228
Cost after epoch 600: 0.313815
Cost after epoch 700: 0.253708
Cost after epoch 800: 0.203900
Cost after epoch 900: 0.166453
Cost after epoch 1000: 0.146636
Cost after epoch 1100: 0.107281
Cost after epoch 1200: 0.086697
Cost after epoch 1300: 0.059341
Cost after epoch 1400: 0.052289
```

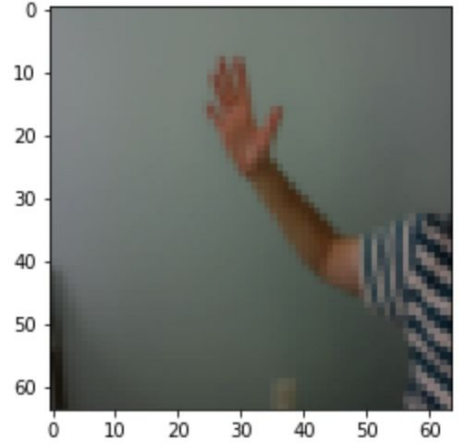



```
Parameters have been trained!
Train Accuracy: 0.9990741
Test Accuracy: 0.725
```

Insights:

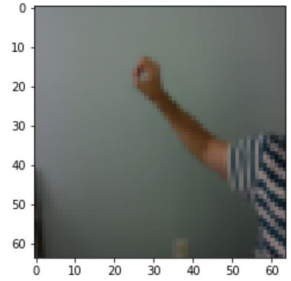
- Your model seems big enough to fit the training set well. However, given the difference between train and test accuracy, you could try to add L2 or dropout regularization to reduce overfitting.
- Think about the session as a block of code to train the model. Each time you run the session on a minibatch, it trains the parameters. In total you have run the session a large number of times (1500 epochs) until you obtained well trained parameters.

```
Your algorithm predicts: y = 2
```

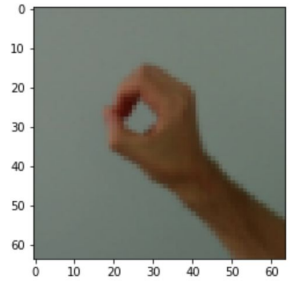


Optimization Algorithms

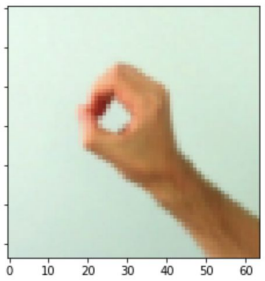
Your algorithm predicts: y = 0



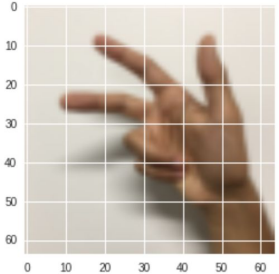
Your algorithm predicts: y = 4



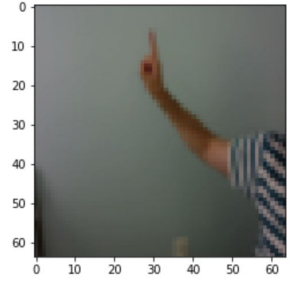
Your algorithm predicts: y = 4



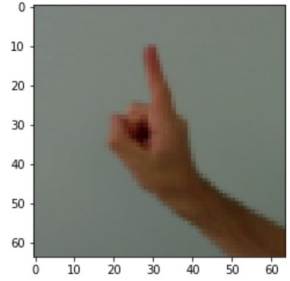
y = 3



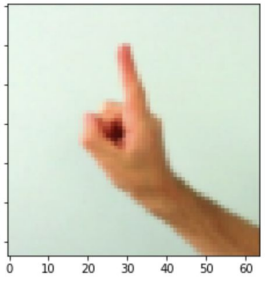
Your algorithm predicts: y = 0



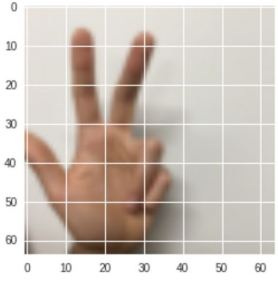
Your algorithm predicts: y = 3



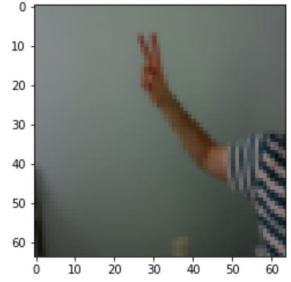
Your algorithm predicts: y = 0



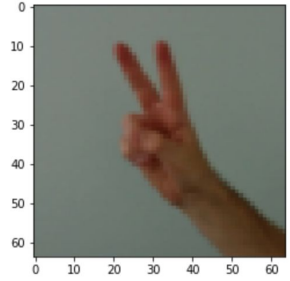
y = 3



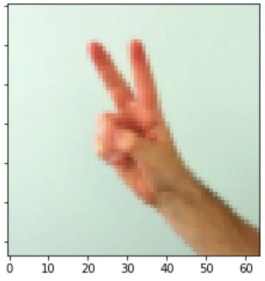
Your algorithm predicts: y = 2



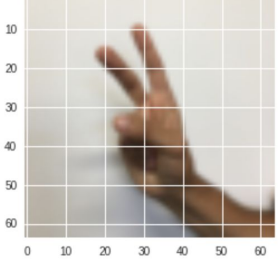
Your algorithm predicts: y = 5



Your algorithm predicts: y = 5



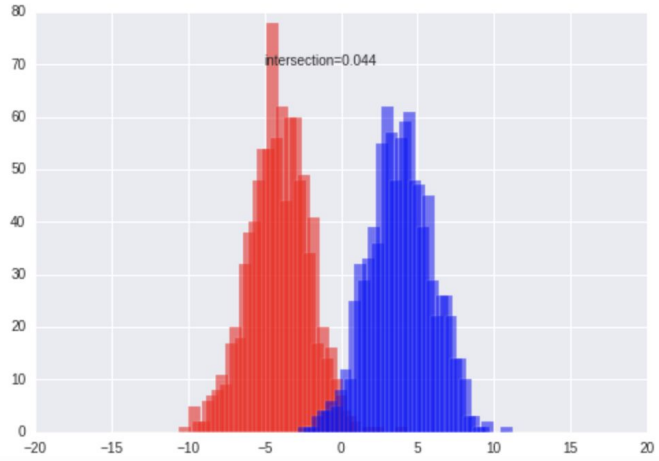
y = 2

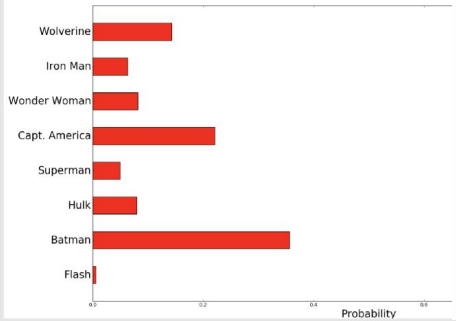


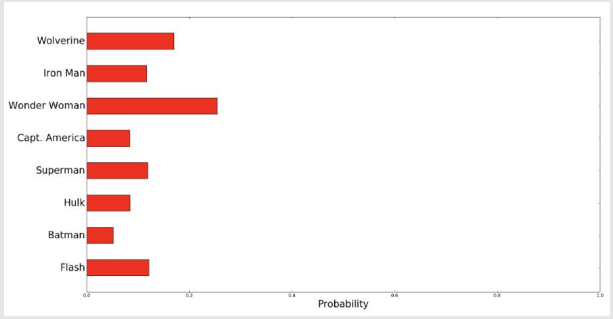
Optimization Algorithms

Superheroes classification

```
histogram_intersection(mu_1=-4, mu_2=4, datapoints=1000, bins=100, range=15, theme='seaborn', alpha=0.5);
```



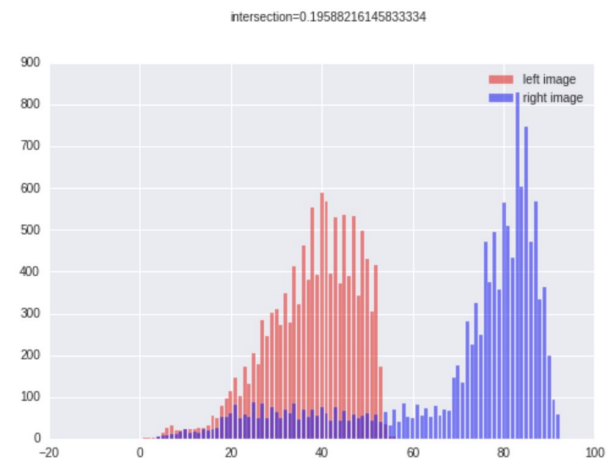


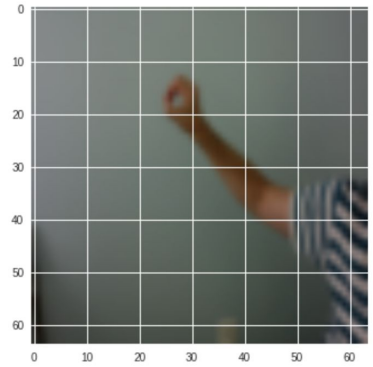


https://mpatacchiola.github.io/blog/2016/11/12/the-simplest-classifier-histogram-intersection.html

Optimization Algorithms

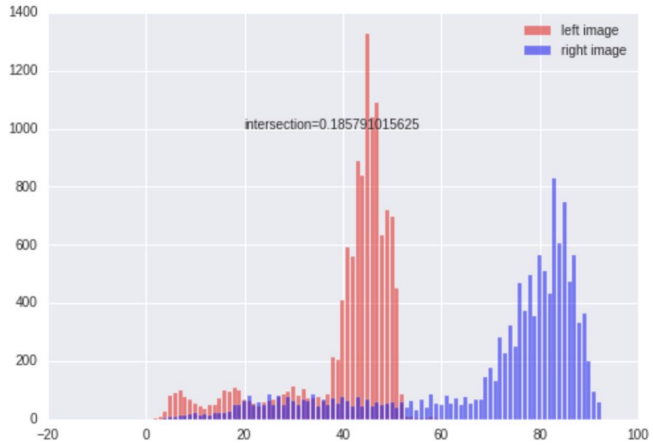
```
compare_images(scaling_factor*image_1, scaling_factor*image_4,
```

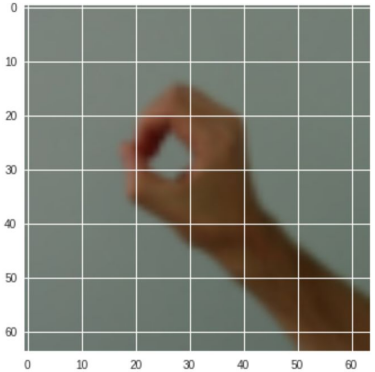




```
(64, 64, 3)
<class 'numpy.ndarray'>
```

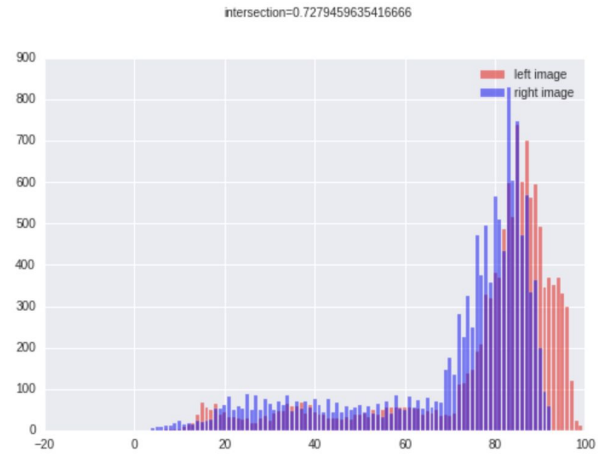
```
compare_images(scaling_factor*image_2, scaling_factor*image_4,
```

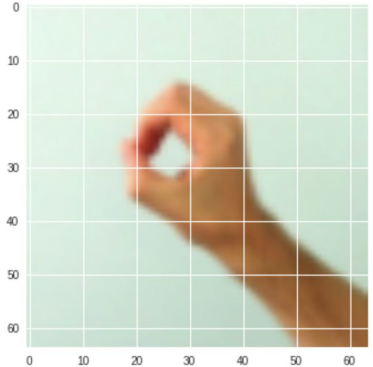




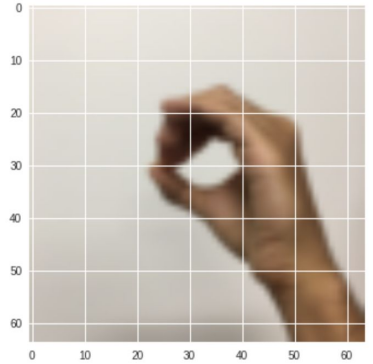
```
(64, 64, 3)
<class 'numpy.ndarray'>
```

```
compare_images(scaling_factor*image_3, scaling_factor*image_4,
```





```
(64, 64, 3)
<class 'numpy.ndarray'>
```



```
(64, 64, 3)
<class 'numpy.ndarray'>
```

Questions

Discussion

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

·$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

·$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

·$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

·$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

·$\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

2 Deep Learning representations

For representations:

· nodes represent inputs, activations or outputs

· edges represent weights or biases

Here are several examples of Standard deep learning representations

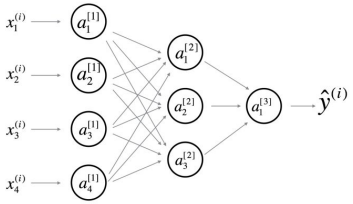


Figure 1: Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc...) that should appear on the edges

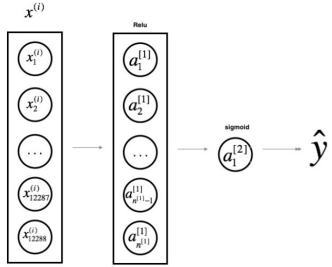


Figure 2: Simplified Network: a simpler representation of a two layer neural network, both are equivalent.