Chapter 8

Practical Considerations in Machine Learning

8.1 Learning Objectives

So far, we have covered the basic principles of regression and neural networks. In practice, typical tasks are much more complicated than our examples presented in previous chapters. Complicated tasks usually require further considerations in various aspects. In this chapter, we will discuss some important practical considerations in deploying machine learning algorithms with an emphasis on neural networks.

No free lunch principle implies that, technically, any algorithm that is good at some problems is bad at others. Thus, applied machine learning is a highly iterative process. Before learning the parameters for a type of model, we have to set up the constraints for the model. For example, to build a L-layer neural network for an image classification, we need to define the number of layers, the number of units in each layer, learning rate, activation functions, and so on. To achieve a relative best result, we usually try different settings, and run the code to see how it works. This *try and see* cycle may repeat iteratively multiple times as shown in Fig.1. Understanding the basic principles of some settings can guide us to debug or improve the model performance.

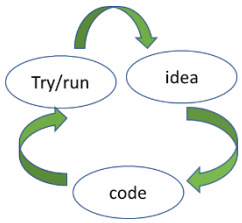


Fig.1 try and see cycle

Specifically, the following concepts will be covered in this chapter:

- dataset, variance and bias balance
- input normalization, weight initialization
- regularization
- mini-batch gradient descent
- Adam optimization
- Gradient check

8.2 Dataset, Variance and Bias

8.2.1 Dataset

Machine learning is driven by data. The availability of big data is essential to for a machine learning model having "*intelligence*". Ideally, all the data are expected to be drawn from the same distribution population. The entire dataset is usually randomly

divided into three parts: training, dev (development) and test sets. It is a common practice to maintain an approximate same categorical ratio in the three subsets, which is called stratified sampling. We train the algorithm using training set, validate using dev set, analyze errors and repeat until error on training and dev sets reduce. We pick the best performing model and use this model to find accuracy on test set. The percentages of three parts are typically 60%, 20% and 20% for training, dev and test sets, respectively. However, if the number of data examples is big (e.g. more than 1,000,000), the percentages for dev and test sets are expected to be reduced. For example, for a size of 1,000,000 dataset, a reasonable partition would be 980,000 (98% training), 10,000 (1% dev), and 10,000 (1% test). Practically, it is ok that the entire data may not be collected from the same population. However, it is important to choose the dev and test sets from the same distribution.

8.2.2 Variance vs Bias

One of the important aspects of error analysis is to identify ***Bias*** and ***Variance***. The value of the error (misclassified examples) on the training set is known as Bias and the difference between the value of error on dev set and value of error on training set is known as Variance. In supervised learning, ***underfitting*** happens when a model is too simple to capture the underlying pattern of the data. These models usually have high bias and low variance. For example, we try to use a linear model to fit a nonlinear dataset. Large prediction errors exit for both training set and dev/testing set. On the contrast, if the model is too complicated and there is no enough data to train the mode, ***overfitting*** will happen. To minimize the training error, the overfitting model intends to capture the noise along with the underlying pattern in data. An overfitting model cannot effectively generalize the prediction for new data examples, thus resulting in a high variance. Fig.2 visualizes these concepts for a curve fitting task.

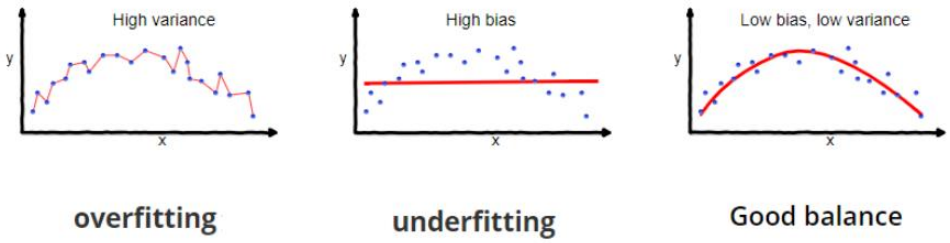


Fig.2 Overfitting/underfitting vs. high variance/high bias

Based on the values of Bias and Variance, we can proceed with different ways. If the values of bias and Variance are balanced within our expectation, we stop the iteration in Fig.1 and go with the testing on test set. This is the usual cycle in deep learning. Consider the following cases:

1) Train set error — 1% and dev set error — 10% means that our model is overfitting train set and not being able to generalize unseen examples. This is called High Variance and can be reduced by introducing regularization or increasing data amount and then training model again.
2) Train set error — 10% and dev set error — 11% means that our model is under fitting train set. This is called High Bias and can be reduced by training with a bigger network or a different neural network architecture.

3) Train set error — 0.5% and dev set error — 1% means that our model is performing well, and we can use this model to test on test set.

A mathematical analysis of bias-variance trade-off can be found in chapter 2 in "*an introduction to statistical learning*". A basic recipe to fix high bias or high variance problems for machine learning is illustrated in Fig.3.

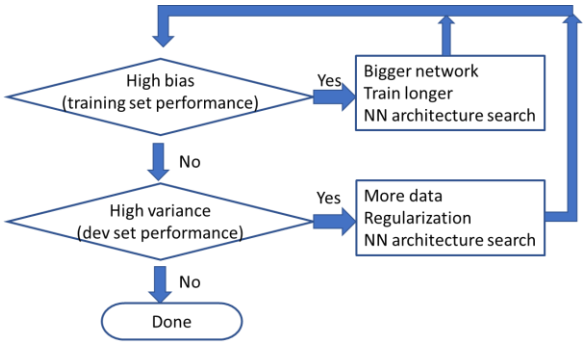


Fig.3 Basic recipe for bias-variance tradeoff

8.3 Input Normalization

In many applications, the features in the data are measured by different scales. For example, the value of feature x_1 may vary in the range from 1 to 1000 while the range of feature x_2 is from 0 to 1. To achieve an efficient optimization computation, it is essential to transform them into quantities with similar scales. It is a common practice to independently normalize each feature into the quantities with zero mean and unit variance. Then the normalized data is used to train the model. To predict on test set, we use the same mean and variance, which was used to normalize training set, to normalize the test set. The prediction will be performed on the normalized test set. The normalized feature of a feature x_i is calculated as:

$$z_i = \frac{x_i - u_i}{s_i} \tag{1.a}$$

where u_i is the mean of the feature x_i, and s_i is the standard deviation of the feature x_i.

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i(j) \tag{1.b}$$

$$s_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}(x_i(j) - \mu_i)^2 \tag{1.c}$$

8.4 W initialization

Training a neural network starts with a set of initialized parameters W and b. It is important to make appropriate initialization on these parameters. In previous chapters, it is known that initializing W to zero does not work. Setting W to zero makes the hidden units symmetric and continues for all the n iterations you run. The resulting neural network is no better than a linear model. It is important to note that setting biases b to 0 will not create any troubles as non-zero W take care of breaking the symmetry and even if bias is 0, the values in every neuron are still different.

Therefore, initializing W randomly is a sounding option. For example, in chapter "*Multiple-Layer Neural Networks*", we noted that the 2-layer model for image classification can be initialized by either

```
parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1]) *0.01
```

or

```
parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])/ np.sqrt(layer_dims[l-1])
```

However, for the L-layer model (L=4, layers_dims = [12288, 20, 7, 5, 1]), the first initialization (*0.01) did not work. The later one was used and worked. Why?

Initializing weights randomly, following standard normal distribution (np.random.randn(size_l, size_l-1) in Python) while working with a (deep) network can potentially lead to 2 issues — vanishing gradients or exploding gradients.

a) *Vanishing gradients* — In case of deep networks, for any activation function, the absolute value of the derivatives (dW) will get smaller and smaller as we go backwards with every layer during back propagation. The earlier layers are the slowest to train in such a case. The weight update is minor and results in slower convergence. This makes the optimization of the loss function slow. In the worst case, this may completely stop the neural network from training further. More specifically, in case of sigmoid(z) and tanh(z), if your weights are large, then the gradient will be vanishingly small, effectively preventing the weights from changing their value. This is because abs(dW) will increase very slightly or possibly get smaller and smaller every iteration. With RELU(z) vanishing gradients are generally not a problem as the gradient is 0 for negative (and zero) inputs and 1 for positive inputs.

b) Exploding gradients — This is the exact opposite of vanishing gradients. Consider you have non-negative and large weights and small activations A (as can be the case for sigmoid(z)). When these weights are multiplied along the layers, they cause a large change in the cost. Thus, the gradients are also going to be large. This means that the changes in W, by W — α * dW, will be in huge steps, the downward moment will increase. This may result in oscillating around the minima or even overshooting the optimum again and again and the model will never learn! Another impact of exploding gradients is that huge values of the gradients may cause number overflow resulting in incorrect computations or introductions of NaN's. This might also lead to the loss taking the value NaN.

Let's consider the linear transformation in a single unit.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b \tag{2}$$

Assume b=0. If n is larger, $\{w_i\}$ should be initialized smaller so that z falls into a range for which the activation g(z) has an appropriate derivative for backward propagation. Thus for a deep neural network, instead of drawing from standard normal distribution, we are drawing W from normal distribution with variance k/n, where k depends on the activation function. While these heuristics do not completely solve the exploding/vanishing gradients issue, they help mitigate it to a great extent. The most common are:

a) For RELU(z) — We multiply the randomly generated values of W by:

$$\sqrt{\frac{2}{n^{[l-1]}}} \tag{3}$$

```
W[l]=np.random.randn(n[l],n[l-1])*np.sqrt(2/n[l-1])
```

b) For tanh(z) — The heuristic is called *Xavier* initialization. It is similar to the previous one, except that k is 1 instead of 2.

$$\sqrt{\frac{1}{n^{[l-1]}}} \tag{4}$$

c) Another commonly used heuristic is:

$$\sqrt{\frac{2}{n^{[l]}+n^{[l-1]}}} \tag{5}$$

8.5 Regularization

A model with high variance is overfitting train set and not being able to generalize unseen examples. This problem can be reduced by introducing regularization or increasing data amount and then training model again. Since collecting more data is usually prohibitive expensive in practice, regularization is often a practical option to address the overfitting issue. We discussed regularization for logistic regression in chapter 5. Here we re-visit regularization with a focus on neural networks.

8.5.1 Regularization for logistic regression

The cost function of logistic regression can be represented by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\ln\left(1-\hat{y}^{(i)}\right)\right] \tag{6}$$

where, $W \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, n_x is the number of features in the input X. The idea of regularization is to modify the cost function by adding a new term to the cost function, that is, to introduce the weight penalty. The new term is proportional to the norm of W. The consequence of this modification is to reduce the magnitude of W, and thus to reduce the sensitivity of the model to data variation.

L2 regularization is given by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|W\|_2^2 \tag{7}$$

$$\|W\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = W^T W$$

L1 regularization is given by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|W\|_1 \tag{8}$$

$$\|W\|_1 = \sum_{j=1}^{n_x} |w_j|$$

where, λ is regularization parameter.

8.5.2 Regularization for neural network

Like the regularization for logistic regression, we can have the regularization for neural network by adding the penalty of weights to cost function,

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_F^2 \tag{9}$$

where $\left\|W^{[l]}\right\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\left(w_{ij}^{[l]}\right)^2$ is called Frobenius norm.

The backward propagation will be modified as

$$dW^{[l]} = dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) + \frac{\lambda}{m}W^{[l]} \tag{10}$$

The parameter update is given by

$$\begin{aligned} W^{[l]} &:= W^{[l]} - \alpha * dW^{[l]} = W^{[l]} - \alpha * dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) - \alpha\frac{\lambda}{m}W^{[l]} \\ &= \left(1 - \alpha\frac{\lambda}{m}\right)W^{[l]} - \alpha * dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) \end{aligned} \tag{11}$$

The coefficient of the first term, $1 - \alpha\frac{\lambda}{m} < 1$, implies that the weight is decaying during the parameter update process. The regularization becomes stronger when increasing λ.

How the regularization can prevent overfitting can be interpreted by examining the curve of tanh(z) in Fig.4. If λ is increased to impose a regularization, $W^{[l]}$ will be reduced. Then

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{12}$$

will be reduced. The activation function $g(Z^{[l]})$ will work in an approximately linear region when $Z^{[l]}$ is small. Thus, the behavior of the whole neural network moves toward the linear direction, which leads to a relatively simple model.

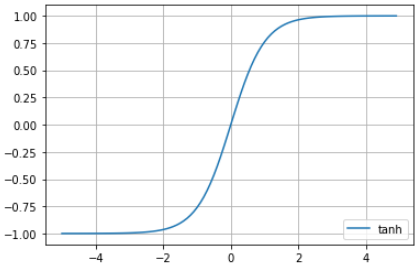


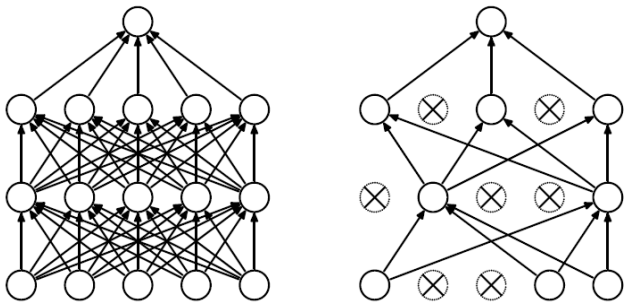
Fig.4 curve of tanh(z)

8.5.3 Dropout for regularization (*optional)

(reference: Dropout: Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever Ruslan Salakhutdinov, A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research* 15 (2014) 1929-1958)

Dropout is another technique to address the overfitting problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Fig.5. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability *p* independent of other units, where *p* can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.



(a) Standard Neural Net (b) After applying dropout.

Fig. 5: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer. A new hyperparameter is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. After each parameter update during the training process, the units to be dropped out will be randomly selected again. The outputs of the retained units will be scaled larger by dividing *p*, to avoid the outputs being varnished after many iterations.

Let's use layer 2 as an example to explain how to implement dropout in forward propagation and backward propagation. Other layers (except layer L) can be done in the same way.

Forward dropout:

```
D2 = np.random.rand(A2.shape[0], A1.shape[1])
D2 = (D2<keep_prob)
A2 = np.multiply(A2,D2)
A2 = A2/keep_prob
```

Backward dropout: the mask matrix D2 generated in forward propagation will be used in backward.

```
    ### start code
        dA2 = np.multiply(dA2, D2)
        dA2 = dA2/keep_prob
        ### end code

        dZ2 = np.multiply(dA2, np.int64(A2 > 0))
```

The elements in A2 associated with the dropped units will be equal to zero. The resulting W will be used for predictions. Dropout is only applied at the training phase. In other words, no dropout will be applied for prediction at test time.

8.6 Mini-batch Gradient Descent

8.6.1 Three types of gradient descent

As discussed in previous chapters, we know that gradient descent is a widely used optimization algorithm for finding the weights or coefficients of machine learning algorithms, such as logistic regression and artificial neural networks. The goal of an optimization algorithm is to find model parameters (e.g. coefficients or weights) that minimize the error of the model on the training dataset. It achieves this by making changes of model parameters that move the error along a gradient or slope of errors down toward a minimum value. This gives the algorithm its name of "gradient descent." Gradient descent can vary in terms of how the data examples are used to update the model. The three main types of gradient descent are **batch, stochastic, and mini-batch**.

Batch gradient descent is the gradient descent algorithm that updates the model parameters each time based on the entire training dataset. So far, all gradient descent algorithms in previous chapters were implemented as batch gradient descent. One cycle through the entire training dataset is called a training **epoch**. Therefore, it is often said that batch gradient descent performs model updates at the end of each training epoch. The major disadvantage of batch gradient descent is that a larger memory is required for one update, and thus leading to a slower update speed.

Stochastic gradient descent, often abbreviated SGD, is a variation of the gradient descent algorithm that calculates the error **and** updates the model for each example in the training dataset. The update of the model for each training example means that stochastic gradient descent is often called an online machine learning algorithm.

Mini-batch gradient descent is another variation of the gradient descent algorithm that splits the training dataset into small batches, and the model parameters are updated each time based on one batch. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning. The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima. The batched updates provide a computationally more efficient process than stochastic gradient descent. The batching allows both the efficiency of not having all training data in memory and algorithm implementations. However, mini-batch requires the configuration of an additional "mini-batch size" hyperparameter for the learning algorithm.

8.6.2 Implementation of mini-batch gradient descent

Suppose that the training dataset has *m=5,000,000* examples $\left(x^{(i)}, y^{(i)}, i = 1,2 \ldots, 5{,}000{,}000\right)$. We can split the dataset into many mini-batches with a particular batch size, say 1000. Thus, there are total 5000 mini-batches with the batch size 1000. The mini-batches are denoted by $\left(X^{\{t\}}, Y^{\{t\}}, t = 1,2, \ldots, 5{,}000\right)$. The batch gradient descent is applied to mini-batches sequentially, which is call mini-batch gradient descent. The algorithm for a neural network with regularization is described as

Repeat {
for t=1,2,…, 5000
{

Forward propagation on $X^{\{t\}}\ Y^{\{t\}}$ (1000 examples)

$Z^{[l]} = W^{[l]}X^{\{t\}} + b^{[l]}$

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

...

$A^{[L]} = g^{[L]}\left(Z^{[L]}\right)$

Compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{t\ batch} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2\cdot 1000}\sum_{l}\left\|W^{[l]}\right\|_F^2$

Backward to compute gradient using $X^{\{t\}}\ Y^{\{t\}}$ (1000 examples)

Update parameters

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

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

}

To understand the behavior of the mini-gradient descent, we can plot the cost J as a function of t. The cost plot should have a similar overall trend as that of the batch gradient descent, but with some noise on the curve, sketched in Fig.6.

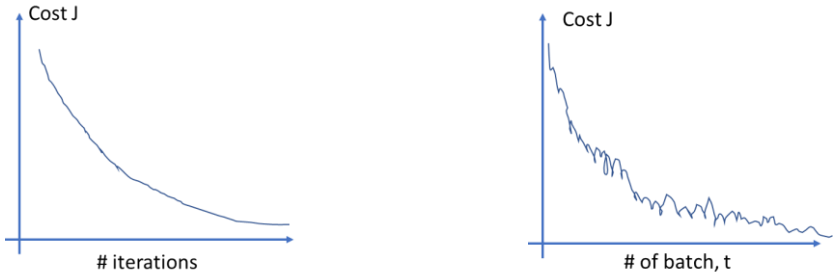


Fig.6 Cost curves. Left: batch gradient descent. Right: mini-gradient descent

8.6.3 Selection of mini-batch size

The mini-batch gradient descent introduces a hyperparameter, mini-size. If the mini-size is equal to the total number of examples in the training set, *m*, then the algorithm is a batch gradient descent. If mini-size =1, the algorithm is a stochastic gradient descent.

In general, small values of mini-size give a learning process that converges quickly at the cost of noise in the training process. Large values give a learning process that converges slowly with accurate estimates of the error gradient. Fig.7 illustrates the converge paths for batch (purple), mini-batch(blue) and stochastic (red) gradient algorithms. For the batch gradient descent, the path (purple) is most straightforward, but each update takes a long time because it involving the entire dataset. For stochastic, the path (red) demonstrates a big noise, but each update takes a small time. The mini-batch path is generally a good balance.

A few practical tips for choosing mini-batch size are suggested as following (based on the current hardware technology in 2020).

1) If the training set is small, e.g. less than 2000, then use batch gradient descent.
2) Typically, mini-batch size is the power of two, such as 2, 4, 16, 32, 64, 128, …. , to match the hardware memory structures.
3) It is a good idea to review learning curves of model validation error against training time with different batch sizes when tuning the batch size.
4) Tune batch size and learning rate **after** tuning all other hyperparameters.

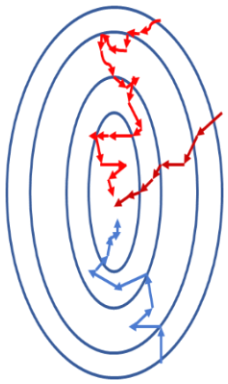


Fig.7 converge path

8.7 Adam Optimization

8.7.1 Gradient descent with momentum

Due to the limited data amount in a mini-batch, there is significant oscillating in the converge path if the parameters are updated only based on the current batch gradients. Furthermore, the steepest gradient descent demonstrates a "*zigzag*" convergence path as shown in Fig.3 in Chapter 2. Fortunately, the overall direction of the converge path eventually points to the minimal cost, shown in Fig.8. Thus, if the average of the parameter update directions over a certain number of previous directions (including current mini-batch direction) is used, the path is expected to be smoother. In other words, the learning speed can be improved by averaging the recent gradients. Since the gradients are becoming more accurate as the iterations on t proceed, the more recent gradients should have larger weights when the averaging is performed. We compute exponentially weighted averages over the gradients of all previous mini-batches.

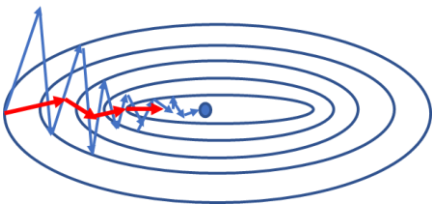


Fig.8 gradient descent with momentum. The red path is obtained by averaging a few previous mini-batch gradients.

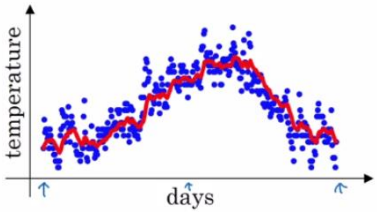


Fig.9 average of temperatures

Now, let's explain the algorithm "*exponentially weighted average*". Consider an example of computing a smooth curve to approximate the temperature in London over a year, given a temperature value of each day, $\{\theta_1, \theta_2, \dots \theta_{181}, \dots\}$, as shown in Fig.9. The exponentially weighted averages, denoted by $\{v_1, v_2, \dots v_{181}, \dots\}$, can be used for this purpose, and computed by

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t \tag{13}$$

where β is a parameter for weight control $(0 < \beta < 1)$, $v_0 = 0$. v_t can be interpreted as the approximate average over $\frac{1}{1-\beta}$ day's temperatures. For example, v_t is the approximate 10-day average if $\beta = 0.9$. Given zero initial value for v_t, (13) can be expressed by $\{\theta_1, \theta_2, \dots \theta_{181}, \dots\}$ explicitly

$$v_t = (1-\beta)(\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \cdots + \beta^{t-1}\theta_1) \tag{14}$$

(14) shows that the weights are exponentially distributed over the past values, and the larger the β, the more weights are put on the far away past values. A minor problem with (13) or (14) is the bias existing in the beginning of the process due to the zero initialization. A **bias correction** is available to compensate this initial bias

$$v_t \coloneqq \frac{v_t}{1-\beta^t} \tag{15}$$

As t increases, the effect of the bias correction will be ignored.

Given $\{\theta_1, \theta_2, \dots \theta_{181}, \dots\}$ and β, the algorithm for exponentially weighted average can be described as follows.

$v_\theta = 0$
Repeat {
    Get next θ_t
    $v_\theta \coloneqq \beta v_\theta + (1-\beta)\theta_t$ (16)
}

Now let's return to gradient descent with **momentum**. On each iteration on t (mini-batches), the parameters W and b are updated based on the exponentially weighted average of gradients of previous mini-batch iterations. The algorithm can be described as

Initialization: $v_{dw} = 0, v_{db} = 0$
On iteration t:
    Compute dW, db on the current mini-batch
    $v_{dw} = \beta v_{dw} + (1-\beta)dW$
    $v_{db} = \beta v_{db} + (1-\beta)db$
    $W \coloneqq W - \alpha v_{dw}$
    $b \coloneqq b - \alpha v_{db}$ (17)

Note that there are two hyperparameters α, β for this update. If $\beta = 0.9$, the model parameters W and b are updated based on the gradient average over the past 10 gradients.

A variation of the momentum algorithm is called **RMS prop** (root means square Propagation), given by

Initialization: $s_{dw} = 0, s_{db} = 0$
On iteration t:

Compute dW, db on the current mini-batch

$$s_{dw} = \beta s_{dw} + (1-\beta)(\|dW\|^2)$$
$$s_{db} = \beta s_{db} + (1-\beta)(\|db\|^2)$$
$$W := W - \alpha \frac{dW}{\sqrt{s_{dw}}+\varepsilon},\ b := b - \alpha \frac{db}{\sqrt{s_{db}}+\varepsilon} \quad (18)$$

8.7.2 Adam optimization algorithm

Adam optimization algorithm is a combination of gradient descent with momentum and RMS prop, an adaptive momentum estimation. It can be described as

Initialization: $v_{dw} = 0, s_{dw} = 0, v_{db} = 0, s_{db} = 0$
On batch iteration t:

Compute dW, db on the current mini-batch

$$v_{dw} = \beta_1 v_{dw} + (1-\beta_1)dw,$$
$$v_{db} = \beta_1 v_{db} + (1-\beta_1)db$$
$$s_{dw} = \beta_2 s_{dw} + (1-\beta_2)(dw * dw),$$
$$s_{db} = \beta_2 s_{db} + (1-\beta_2)(db * db)$$
$$v_{dw} := \frac{v_{dw}}{1-\beta_1^t}$$
$$v_{db} := \frac{v_{db}}{1-\beta_1^t}$$
$$s_{dw} := \frac{s_{dw}}{1-\beta_2^t}$$
$$s_{db} := \frac{s_{db}}{1-\beta_2^t}$$
$$W := W - \alpha \frac{v_{dw}}{\sqrt{s_{dw}}+\varepsilon},\ b := b - \alpha \frac{v_{db}}{\sqrt{s_{db}}+\varepsilon} \quad (19)$$

Typical hyperparameters:

α: needs to be tuned
β_1: 0.9 for dW, db
β_2: 0.999 for dW*dW, db*db
ε: 10E-8 (avoid dividing by zero)

In practice, Adam optimization algorithm has been implemented in packages of popular program libraries.

8.7.3 Learning rate decay

The hyperparameter α, called learning rate, defines the step size of model parameter updating. It is desirable to have relatively large learning rates at the earlier updating, while smaller learning rates when the updating is approaching close to the minimal cost. In other words, we would like the learning rate to decay with the update process. We define 1 epoch as one pass through the dataset, as shown in Fig.10.

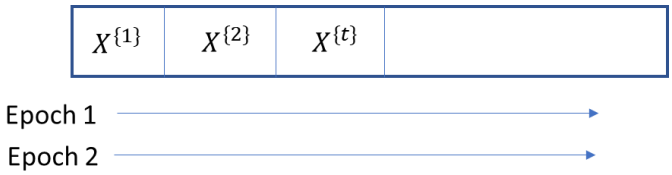


Fig.10 Epoch concept

An empirical learning rate decay is given by

$$\alpha = \frac{\alpha_0}{1+decay\_rate*epoch\_num} \tag{20}$$

For example, $\alpha_0 = 0.2$, decay_rate=1, then

EPOCH	α
1	0.1
2	0.067
3	0.05
4	0.04

Other learning rate decay equations include

$$\alpha = 0.95^{epoch\_num} \cdot \alpha_0 \tag{21}$$

$$\alpha = \frac{k}{\sqrt{epoch\_num}}\alpha_0 \text{ or } \alpha = \frac{k}{\sqrt{t}}\alpha_0 \tag{22}$$

8.8 Gradient checking

The gradient computation is usually difficult to debug. We describe a method for numerically checking the computed derivatives to make sure that the derivatives were correctly calculated. Carrying out the derivative checking procedure will significantly increase your confidence in the correctness of your code, though it is required for model training.

Suppose we want to minimize $J(\theta)$. For simplicity, suppose $J:R\mapsto R$, so that $\theta\in R$. If we are using an optimization algorithm, then we usually have implemented some function g(θ) that computes $\frac{d}{d\theta}J(\theta)$.

How can we check if our implementation of g is correct? Recall the mathematical definition of the derivative as:

$$\frac{d}{d\theta}J(\theta) = \lim_{\varepsilon\to 0}\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \tag{23}$$

Thus, at any specific value of θ, we can numerically approximate the derivative as follows:

$$\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \tag{24}$$

In practice, we set ε to a small constant, say around 10^{-4}. Thus, given a function g(θ) that is supposedly computing $\frac{d}{d\theta}J(\theta)$, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \tag{25}$$

The degree to which these two values should approximate each other will depend on the details of *J*. But assuming $\varepsilon = 10^{-4}$, we usually find that the left- and right-hand sides of (25) will agree to at least 4 significant digits (and often many more).

Now, consider the case where $\theta \in R^n$ is a vector rather than a single real number (so that we have *n* parameters that we want to learn), and $J: R^n \mapsto R$. We now generalize our derivative checking procedure to the case where θ is a vector. If ever we are optimizing over several variables or over matrices, we can always pack these parameters into a long vector and use the same method here to check our derivatives. (This will often need to be done anyway if we want to use off-the-shelf optimization packages.)

Suppose we have a function $g_i(\theta)$ that purportedly computes $\frac{\partial}{\partial\theta_i}J(\theta)$; we'd like to check if g_i is outputting correct derivative values. Let $\theta^{(i+)} = \theta + \varepsilon \times \overrightarrow{e_i}$, where

$$\overrightarrow{e_i} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{26}$$

$\overrightarrow{e_i}$ is the i-th basis vector (a vector of the same dimension as θ, with a "1" in the i-th position and "0"s everywhere else). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ε. Similarly, let $\theta^{(i-)} = \theta - \varepsilon \times \overrightarrow{e_i}$, be the corresponding vector with the i-th element decreased by ε. We can now numerically verify $g_i(\theta)$ correctness by checking, for each *i*, that:

$$g_i(\theta) \approx g_{approx,i}(\theta) = \frac{J(\theta^{(i+)})-J(\theta^{(i-)})}{2\varepsilon} \tag{27}$$

In practice, we can calculate the relative distance between $g(\theta)$ and $g_{approx}(\theta)$, where $g(\theta)$ is a vector consisting of $g_i(\theta)$ and $g_{approx}(\theta)$ is a vector consisting of $g_{approx,i}(\theta)$, *i=1,2…, n*.

$$d = \frac{\left\| g_{approx}(\theta) - g(\theta) \right\|_2}{\left\| g_{approx}(\theta) \right\|_2 + \| g(\theta) \|_2}$$

If d is small enough, say 10^{-7}, then g(θ) is likely correct.

A few reminders are suggested for gradient checking:

- Don't use it in training, only to debug. $g_{approx,i}(\theta)$ is very computationally expensive, so turn off the gradient checking during the training phase.
- If algorithm fails gradient check, look at components to try to identify bug.
- Remember regularization
- Doesn't work with dropout

1) Turn off dropout (keep_prob=1)
2) Use gradient check
3) Turn on dropout (for example, keep_prob=0.8)

8.9 Examples in Python

In this section, we will practice the techniques discussed in this chapter through a few examples.

8.9.1 Parameter initialization

The following codes show the parameter initialization in three different ways: 1) zero; 2) random; and 3) equation (3), where L is the total number of layers.

```
def initialize_parameters_zeros(layers_dims):

    parameters = {}
    L = len(layers_dims)

    for l in range(1,L):
        parameters['W'+str(l)]=np.zeros((layers_dims[l], layers_dims[l-1]))
        parameters['b'+str(l)]=np.zeros((layers_dims[l],1))

    return parameters

def initialize_parameters_random(layers_dims):

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims)

    for l in range(1,L):
        parameters['W'+str(l)]=np.random.randn(layers_dims[l], layers_dims[l-
1])*1.0
        parameters['b'+str(l)]=np.zeros((layers_dims[l],1))

    return parameters

def initialize_parameters_he(layers_dims):

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims)

    for l in range(1,L):
        parameters['W'+str(l)]=np.random.randn(layers_dims[l], layers_dims[l-
1])*np.sqrt(2.0/layers_dims[l-1])
        parameters['b'+str(l)]=np.zeros((layers_dims[l],1))

    return parameters
```

8.9.2 A 3-layer network with regularization and dropout

Now we will develop a 3-layer neural network with regularization or dropout control. The network size is layers_dims =[X.shape[0], 20, 5, 1], shown in Fig.11. Layer 0 is input. Layer 1

(hidden layer) has 20 units with ReLU activations. Layer 2 (hidden layer) has 5 units with ReLU activations. The output layer has one unit with sigmoid activation. Thus this network can perform a binary classification.

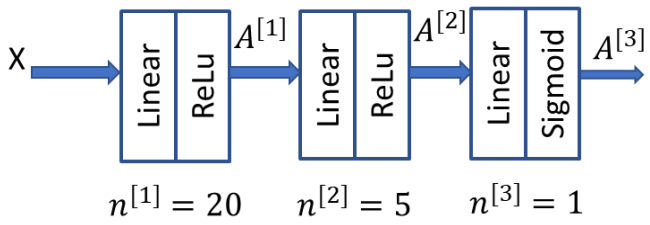


Fig.11 The implemented 3-layer network

The following ch8_utils.py contains some basic functions which will be used later in the main (top-level) file. These basic functions include:

Basic functions for neural networks

```
sigmoid(x)

relu(x)

initialize_parameters(layer_dims)

forward_propagation(X, parameters):

backward_propagation(X, Y, cache):

update_parameters(parameters, grads, learning_rate):

compute_cost(a3, Y):
```

dataset functions

```
load_dataset():

load_planar_dataset(randomness, seed):

load_planar_dataset(seed):

load_2D_dataset():
```

prediction functions

```
predict(X, y, parameters): # for accuracy compute

predict_dec(parameters, X):   # for decision boundary

plot_decision_boundary(model, X, y):
```

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jun  9 16:46:05 2019
ch8_utils.py
@author: weido
"""

import numpy as np
import matplotlib.pyplot as plt
import h5py
```

```
import sklearn
import sklearn.datasets
import sklearn.linear_model
import scipy.io

def sigmoid(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- sigmoid(x)
    """
    s = 1/(1+np.exp(-x))
    return s

def relu(x):
    """
    Compute the relu of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- relu(x)
    """
    s = np.maximum(0,x)

    return s

def load_planar_dataset(seed):

    np.random.seed(seed)

    m = 400 # number of examples
    N = int(m/2) # number of points per class
    D = 2 # dimensionality
    X = np.zeros((m,D)) # data matrix where each row is a single example
    Y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blue)
    a = 4 # maximum ray of the flower

    for j in range(2):
        ix = range(N*j,N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # theta
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        Y[ix] = j

    X = X.T
    Y = Y.T

    return X, Y

def initialize_parameters(layer_dims):
    """
    Arguments:
    layer_dims -
  - python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -
  - python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
```

```
                    W1 -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    b1 -- bias vector of shape (layer_dims[l], 1)
                    Wl -- weight matrix of shape (layer_dims[l-1], layer_dims[l])
                    bl -- bias vector of shape (1, layer_dims[l])

    Tips:
    - For example: the layer_dims for the "Planar Data classification model" would have been [2,2,1].
    This means W1's shape was (2,2), b1 was (1,2), W2 was (2,1) and b2 was (1,1). Now you have to generalize it!
    - In the for loop, use parameters['W' + str(l)] to access Wl, where l is the iterative integer.
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) / np.sqrt(layer_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters

def forward_propagation(X, parameters):
    """
    Implements the forward propagation (and computes the loss) presented in Figure 2.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    parameters -
    - python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3":
                    W1 -- weight matrix of shape ()
                    b1 -- bias vector of shape ()
                    W2 -- weight matrix of shape ()
                    b2 -- bias vector of shape ()
                    W3 -- weight matrix of shape ()
                    b3 -- bias vector of shape ()

    Returns:
    loss -- the loss function (vanilla logistic loss)
    """

    # retrieve parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

```

```
    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)

    return A3, cache

def backward_propagation(X, Y, cache):
    """
    Implement the backward propagation presented in figure 2.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat)
    cache -- cache output from forward_propagation()

    Returns:
    gradients -
     - A dictionary with the gradients with respect to each parameter, activation and
      pre-activation variables
    """
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

    dZ3 = A3 - Y
    dW3 = 1./m * np.dot(dZ3, A2.T)
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T)
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T)
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                 "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients

def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(i)] = Wi
                    parameters['b' + str(i)] = bi
    grads -- python dictionary containing your gradients for each parameters:
                    grads['dW' + str(i)] = dWi
                    grads['db' + str(i)] = dbi
    learning_rate -- the learning rate, scalar.
```

```
179.
180.     Returns:
181.     parameters -- python dictionary containing your updated parameters
182.     """
183.
184.     n = len(parameters) // 2 # number of layers in the neural networks
185.
186.     # Update rule for each parameter
187.     for k in range(n):
188.         parameters["W" + str(k+1)] = parameters["W" + str(k+1)] - learning_rate *
        grads["dW" + str(k+1)]
189.         parameters["b" + str(k+1)] = parameters["b" + str(k+1)] - learning_rate *
       grads["db" + str(k+1)]
190.
191.     return parameters
192.
193. def predict(X, y, parameters):
194.     """
195.     This function is used to predict the results of a  n-layer neural network.
196.
197.     Arguments:
198.     X -- data set of examples you would like to label
199.     parameters -- parameters of the trained model
200.
201.     Returns:
202.     p -- predictions for the given dataset X
203.     """
204.
205.     m = X.shape[1]
206.     p = np.zeros((1,m), dtype = np.int)
207.
208.     # Forward propagation
209.     a3, caches = forward_propagation(X, parameters)
210.
211.     # convert probas to 0/1 predictions
212.     for i in range(0, a3.shape[1]):
213.         if a3[0,i] > 0.5:
214.             p[0,i] = 1
215.         else:
216.             p[0,i] = 0
217.
218.     # print results
219.
220.     #print ("predictions: " + str(p[0,:]))
221.     #print ("true labels: " + str(y[0,:]))
222.     print("Accuracy: "  + str(np.mean((p[0,:] == y[0,:]))))
223.
224.     return p
225.
226. def compute_cost(a3, Y):
227.     """
228.     Implement the cost function
229.
230.     Arguments:
231.     a3 -- post-activation, output of forward propagation
232.     Y -- "true" labels vector, same shape as a3
233.
234.     Returns:
235.     cost - value of the cost function
236.     """
237.     m = Y.shape[1]
```

```
238.
239.     logprobs = np.multiply(-np.log(a3),Y) + np.multiply(-np.log(1 - a3), 1 - Y)
240.     cost = 1./m * np.nansum(logprobs)
241.
242.     return cost
243.
244. def load_dataset():
245.     train_dataset = h5py.File('datasets/train_catvnoncat.h5', "r")
246.     train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set
       features
247.     train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set
       labels
248.
249.     test_dataset = h5py.File('datasets/test_catvnoncat.h5', "r")
250.     test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set fea
      tures
251.     test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set lab
      els
252.
253.     classes = np.array(test_dataset["list_classes"][:]) # the list of classes
254.
255.     train_set_y = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
256.     test_set_y = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))
257.
258.     train_set_x_orig = train_set_x_orig.reshape(train_set_x_orig.shape[0], -
      1).T
259.     test_set_x_orig = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T
260.
261.     train_set_x = train_set_x_orig/255
262.     test_set_x = test_set_x_orig/255
263.
264.     return train_set_x, train_set_y, test_set_x, test_set_y, classes
265.
266.
267. def predict_dec(parameters, X):
268.     """
269.     Used for plotting decision boundary.
270.
271.     Arguments:
272.     parameters -- python dictionary containing your parameters
273.     X -- input data of size (m, K)
274.
275.     Returns
276.     predictions -- vector of predictions of our model (red: 0 / blue: 1)
277.     """
278.
279.     # Predict using forward propagation and a classification threshold of 0.5
280.     a3, cache = forward_propagation(X, parameters)
281.     predictions = (a3>0.5)
282.     return predictions
283.
284. def load_planar_dataset(randomness, seed):
285.
286.     np.random.seed(seed)
287.
288.     m = 50
289.     N = int(m/2) # number of points per class
290.     D = 2 # dimensionality
291.     X = np.zeros((m,D)) # data matrix where each row is a single example
292.     Y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blue)
293.     a = 2 # maximum ray of the flower
```

```
294.
295.     for j in range(2):
296.
297.         ix = range(N*j,N*(j+1))
298.         if j == 0:
299.             t = np.linspace(j, 4*3.1415*(j+1),N) #+ np.random.randn(N)*randomness
        # theta
300.             r = 0.3*np.square(t) + np.random.randn(N)*randomness # radius
301.         if j == 1:
302.             t = np.linspace(j, 2*3.1415*(j+1),N) #+ np.random.randn(N)*randomness
        # theta
303.             r = 0.2*np.square(t) + np.random.randn(N)*randomness # radius
304.
305.         X[ix] = np.c_[r*np.cos(t), r*np.sin(t)]
306.         Y[ix] = j
307.
308.     X = X.T
309.     Y = Y.T
310.
311.     return X, Y
312.
313. def plot_decision_boundary(model, X, y):
314.     # Set min and max values and give it some padding
315.     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
316.     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
317.     h = 0.01
318.     # Generate a grid of points with distance h between them
319.     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

320.     # Predict the function value for the whole grid
321.     Z = model(np.c_[xx.ravel(), yy.ravel()])
322.     Z = Z.reshape(xx.shape)
323.     # Plot the contour and training examples
324.     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
325.     plt.ylabel('x2')
326.     plt.xlabel('x1')
327.     plt.scatter(X[0, :], X[1, :], c=y.ravel(), cmap=plt.cm.Spectral)
328.     plt.show()
329.
330. def load_2D_dataset():
331.     data = scipy.io.loadmat('data.mat')
332.     train_X = data['X'].T
333.     train_Y = data['y'].T
334.     test_X = data['Xval'].T
335.     test_Y = data['yval'].T
336.
337.     plt.scatter(train_X[0, :], train_X[1, :], c=train_Y.ravel(), s=40, cmap=plt.c
      m.Spectral);
338.
339.     return train_X, train_Y, test_X, test_Y
```

The following main file implements regularization and dropout functions, and integrates all required functions to train a 3-layer neural network with regularization (λ) or dropout probability control.

```
1.     # -*- coding: utf-8 -*-
2.     """
3.     Created on Sun Jun  9 16:49:00 2019
4.     ch8_1.py
```

```
    @author: weido
    """

    import numpy as np
    import matplotlib.pyplot as plt
    from ch8_utils import sigmoid, relu, plot_decision_boundary, initialize_parameters, load_2D_dataset
    from ch8_utils import predict_dec, compute_cost, predict, forward_propagation, backward_propagation, update_parameters
    import sklearn
    import sklearn.datasets
    import scipy.io

    plt.rcParams['figure.figsize']=(7.0, 4.0)
    plt.rcParams['image.interpolation']='nearest'
    plt.rcParams['image.cmap']='gray'

    # load dataset circles
    train_X, train_Y, test_X, test_Y = load_2D_dataset()

    plt.scatter(train_X[0, :], train_X[1, :],c=train_Y[0,:], s=40, cmap=plt.cm.Spectral)
    plt.xlabel("X[0,:]")
    plt.ylabel("X[1,:]")
    plt.title("data visualization, red: y=0, blue: y=1")
    plt.show()

    def compute_cost_with_regularization(A3, Y, parameters, lambd):
        """
        """
        m = Y.shape[1]
        W1 = parameters["W1"]
        W2 = parameters["W2"]
        W3 = parameters["W3"]

        cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

        ### START CODE HERE ### (approx. 1 line)
        L2_regularization_cost = (1. / m)*(lambd / 2) * (np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))

        cost = cross_entropy_cost + L2_regularization_cost

        return cost

    def backward_propagation_with_regularization(X, Y, cache, lambd):
        """
        Implements the backward propagation of our baseline model to which we added an L2 regularization.

        Arguments:
        X -- input dataset, of shape (input size, number of examples)
        Y -- "true" labels vector, of shape (output size, number of examples)
        cache -- cache output from forward_propagation()
        lambd -- regularization hyperparameter, scalar

        Returns:
        gradients -
- A dictionary with the gradients with respect to each parameter, activation and pre-activation variables
```

```
        """

        m = X.shape[1]
        (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

        dZ3 = A3 - Y

        ### START CODE HERE ### (approx. 1 line)
        dW3 = 1./m * (np.dot(dZ3, A2.T) + lambd * W3)
        ### END CODE HERE ###
        db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

        dA2 = np.dot(W3.T, dZ3)
        dZ2 = np.multiply(dA2, np.int64(A2 > 0))
        ### START CODE HERE ### (approx. 1 line)
        dW2 = 1./m * (np.dot(dZ2, A1.T) + lambd * W2 )
        ### END CODE HERE ###
        db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
        dA1 = np.dot(W2.T, dZ2)
        dZ1 = np.multiply(dA1, np.int64(A1 > 0))
        ### START CODE HERE ### (approx. 1 line)
        dW1 = 1./m * (np.dot(dZ1, X.T) + lambd * W1 )
        ### END CODE HERE ###
        db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

        gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
                     "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
                     "dZ1": dZ1, "dW1": dW1, "db1": db1}

        return gradients

    def forward_propagation_with_dropout(X, parameters, keep_prob):
        """
        Implements the forward propagation

        Arguments:
        X -- input dataset, of shape (input size, number of examples)
        parameters -
    - python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3
    ":
                        W1 -- weight matrix of shape ()
                        b1 -- bias vector of shape ()
                        W2 -- weight matrix of shape ()
                        b2 -- bias vector of shape ()
                        W3 -- weight matrix of shape ()
                        b3 -- bias vector of shape ()
        keep_prob - probability of keeping a neuron active during dropout, scalar
        Returns:
        A3 -
    - last activation value, output of the forward propagation, of shape(1,1)
        cache -- tuple, information stored for computing the backward propagation
        """
        np.random.seed(4)
        # retrieve parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]
        W3 = parameters["W3"]
        b3 = parameters["b3"]

```

```
        # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
        Z1 = np.dot(W1, X) + b1
        A1 = relu(Z1)
        ### start code here ###
        D1 = np.random.rand(A1.shape[0], A1.shape[1])
        D1 = (D1<keep_prob)
        A1 = np.multiply(A1,D1)
        A1 = A1/keep_prob
        ##E end code
        Z2 = np.dot(W2, A1) + b2
        A2 = relu(Z2)


         ### start code here ###
        D2 = np.random.rand(A2.shape[0], A1.shape[1])
        D2 = (D2<keep_prob)
        A2 = np.multiply(A2,D2)
        A2 = A2/keep_prob
        ##E end code


        Z3 = np.dot(W3, A2) + b3
        A3 = sigmoid(Z3)

        cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)

        return A3, cache

    def backward_propagation_with_dropout(X, Y, cache, keep_prob):
        """
        Implement the backward propagation to which we added dropout

        Arguments:
        X -- input dataset, of shape (input size, number of examples)
        Y -- true "label" vector (containing 0 if cat, 1 if non-cat)
        cache -- cache output from forward_propagation()
        keep_prob - probability of keeping a neuron active during dropout, scalar
        Returns:
        gradients -
    - A dictionary with the gradients with respect to each parameter, activation and
     pre-activation variables
        """
        m = X.shape[1]
        (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache

        dZ3 = A3 - Y
        dW3 = 1./m * np.dot(dZ3, A2.T)
        db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

        dA2 = np.dot(W3.T, dZ3)

        ### start code
        dA2 = np.multiply(dA2, D2)
        dA2 = dA2/keep_prob
        ### end code

        dZ2 = np.multiply(dA2, np.int64(A2 > 0))
        dW2 = 1./m * np.dot(dZ2, A1.T)
        db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

        dA1 = np.dot(W2.T, dZ2)
```

```
        ### start code
        dA1 = np.multiply(dA1, D1)
        dA1 = dA1/keep_prob
        #### end code
        dZ1 = np.multiply(dA1, np.int64(A1 > 0))
        dW1 = 1./m * np.dot(dZ1, X.T)
        db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

        gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                     "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                     "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

        return gradients


    def model(X,Y, learning_rate=0.3, num_iterations=30000, print_cost= True, lambd=
    0, keep_prob=1):

        grads={}
        costs=[]
        m=X.shape[1]
        layers_dims = [X.shape[0],20,5,1]

        #initialize parameters dictionary.


        parameters = initialize_parameters(layers_dims)

        # loop (gradient descent)

        for i in range (0, num_iterations):

            # Linear-->Relu-->linear-->Relu-->linear-->sigmoid
            if keep_prob == 1:
                a3, cache = forward_propagation(X, parameters)
            elif keep_prob <1:
                a3, cache = forward_propagation_with_dropout(X, parameters, keep_pro
    b)

            #cost
            if lambd == 0:
                cost = compute_loss(a3, Y)
            else:
                cost = compute_cost_with_regularization(a3,Y,parameters,lambd)

            #backward propagation
            assert(lambd == 0 or keep_prob ==1)  # No reg and dropout at same time
            if lambd ==0 and keep_prob ==1:
                grads=backward_propagation(X,Y, cache)
            elif lambd != 0:
                grads=backward_propagation_with_regularization(X,Y, cache, lambd)
            elif keep_prob < 1:
                grads=backward_propagation_with_dropout(X,Y, cache, keep_prob)


            parameters = update_parameters(parameters, grads, learning_rate)

            if print_cost and i % 1000 ==0:
                print("cost after iteration {}: {}".format(i,cost))
                costs.append(cost)

```

```
        #plot the loss
        plt.plot(costs)
        plt.ylabel('cost')
        plt.xlabel('iterations (per hundreds)')
        plt.title("learning rate = " + str(learning_rate))
        plt.show()

        return parameters

    parameters = model(train_X, train_Y, lambd=0, keep_prob = 0.8)
    print("on the train set:")
    predictions_train = predict(train_X, train_Y, parameters)

    print("on the test set:")
    predictions_train = predict(test_X, test_Y, parameters)

    plt.title("Model with dropout")
    #axes = plt.gca
    plt.xlim([-0.75, 0.40])
    plt.ylim([-0.75, 0.65])
    plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```

Remarks: this file does not allow to apply weight regularization and dropout at the same time. In other words, only one of them can be applied for a particular design.

```
assert(lambd == 0 or keep_prob ==1)
```

The training data examples are generated from load_2D_dataset() in scipy.io, shown in Fig.12. `parameters = model(train_X, train_Y, lambd=0, keep_prob = 1)` is set for no weight regularization and no dropout. The result is overfitting (high variance) shown in Fig.13.

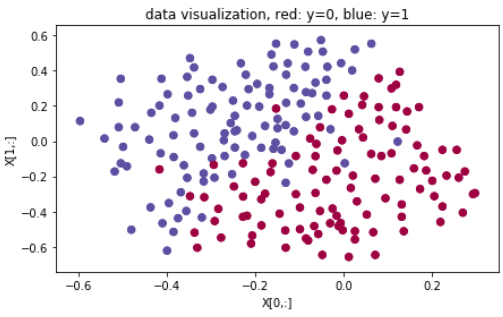


Fig.12 training examples

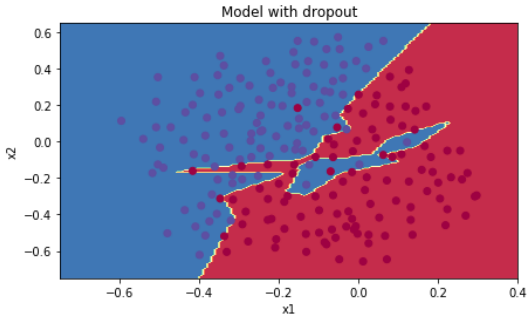
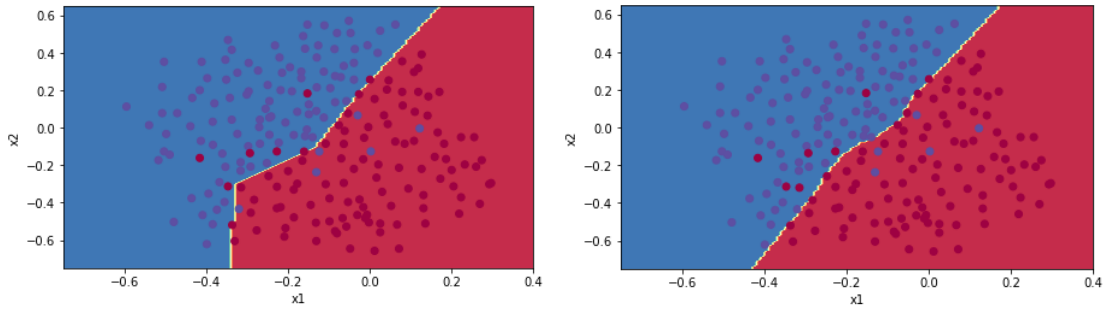


Fig. 13 no regularization and no dropout

Accuracy on training set: 0.976303317535545
Accuracy on test set: 0.92

The statement `parameters = model(train_X, train_Y, lambd=0.1, keep_prob = 1)` implements regularization with λ=0.1, shown in Fig.14(a). Accuracy on training set is 0.9383886255924171

and 0.95 on test set. `parameters = model(train_X, train_Y, lambd=0, keep_prob = 0.8)` implements a dropout with keep probability of 0.8, shown in Fig.14(b). Accuracy is 0.943127962085308 on training set and 0.935 on test set.



a) Weight regularization with λ=0.1 b) dropout with p=0.8

Fig.14 Regularization a) weight regularization; b) dropout

Files:
\machine_learning\NN_settingup\
ch8_utils.py, ch8_1.py

Summary

In this chapter, we present some important considerations in deploying a neural network. These considerations include variance, input normalization, parameter initialization, weight regularization, dropout for regularization, mini-batch gradient descent, optimization methods (momentum and Adam), and gradient check. It should be emphasized that some of them are heuristic (not formally proven) but very helpful. Examples show the details of implementation of regularizations (including dropout) through python programming.

A comprehensive tutorial project on mini-batch gradient descent and optimization methods will be given as a separately chapter (see the next chapter).

References

[1] "Neural networks and deep learning" online course at www.coursera.org

Exercises

1. Parameter initialization. Three heuristic initialization methods are defined by

   a) For RELU(z) — We multiply the randomly generated values of W by:

$$\sqrt{\frac{2}{n^{[l-1]}}}$$

$$W^{[1]}=np.random.randn(n^{[1]},n^{[1-1]})*np.sqrt(2/n^{[1-1]})$$

   b) For tanh(z) — The heuristic is called *Xavier* initialization. It is similar to the previous one, except that k is 1 instead of 2.

$$\sqrt{\frac{1}{n^{[l-1]}}}$$

   c) Another commonly used heuristic is:

$$\sqrt{\frac{2}{n^{[l]}+n^{[l-1]}}}$$

For each method, write a Python function to implement the parameter initialization, given the size of network *layers_dims*. The template of code is given below.

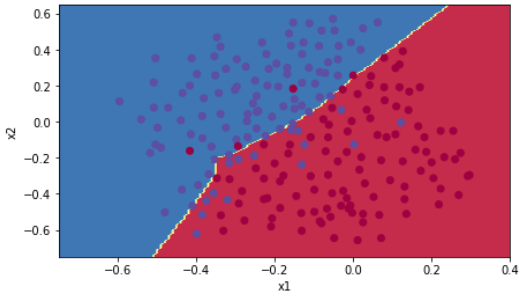
```
def initialize_parameters_xxx(layers_dims):

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims)

    for l in range(1,L):
        parameters['W'+str(l)]=
        parameters['b'+str(l)]=
    return parameters
```

2. In file ch8_1.py, regularization and dropout cannot be applied simultaneously. Please define and complete the following function, which considers both weight regularization and dropout.

```
def backward_propagation_with_regularization_dropout(X, Y, cache, lambd, keep_prob):

  # start your code here

   gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

   return gradients
```

Then in `model(X,Y, learning_rate=0.3, num_iterations=30000, print_cost= True, lambd=0, keep_prob=1)`, include the situation for "lambd ==!0 and keep_prob<1" to call function `backward_propagation_with_regularization_dropout(X, Y, cache, lambd, keep_prob)`.

Your modified ch8_1.py should be able to do “weight regularization and dropout” simultaneously. Try the case “λ=0.1 and keep_prob=0.9”, what are the accuracies on training set and testing set of datasets generated by load_2D_dataset()? (answer: ch8_ex2.py in \machine_learning\NN_setting\, 0.938xx, and 0.93)



(answer plot)

3. Add gradient check to ch8_1.py.