Chapter 6

Practical Considerations in Neural Networks

So far, we have covered the basic principles of regression and neural networks. In practice, many tasks are much more complicated than our examples presented in previous chapters. We usually need to go through iterative design cycles for developing a large neural network, as shown in Fig.6.1. Before training the model, we have to set up the constraints for the model. For example, to build an 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 an acceptable 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.

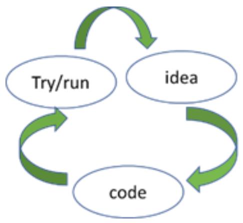


Fig.6.1 try and see cycle

In this chapter we will describe the general architecture of a neural network with an arbitrary layer length L, by mathematical equations for forward propagation and gradient backward propagation. An activation function, called ReLU, is typically used for the neural network hidden layers with a big number of layers. Then we will address some basic practical considerations for designing and training the multiple-layer neural networks. These considerations include normalization, weight initialization, regularization, batch gradient descent, and Adam optimization. The purpose of these considerations is to improve the effectiveness and efficiency of training neural networks.

In this chapter, you will learn:

- Mathematical representation of multiple-layer neural networks
- Overfitting and underfitting
- Regularization techniques: weight penalty and dropout
- Weight initialization
- Mini-batch gradient descent and epoch schedule
- Batch normalization
- Learning rate decay
- Derivative checking

6.1 Multiple-Layer Neural Networks

6.1.1 Architecture

A deep neural network consists of multiple hidden layers with multiple units (or neurons) in each layer. Fig.6.2 shows a fully connected 3-hidden-layer neural network. There are 9 units in each hidden layer. Please note that different hidden layers may have different numbers of units. “Fully-connected” means that the output of any unit in a hidden layer is connected to all units in the next layer.

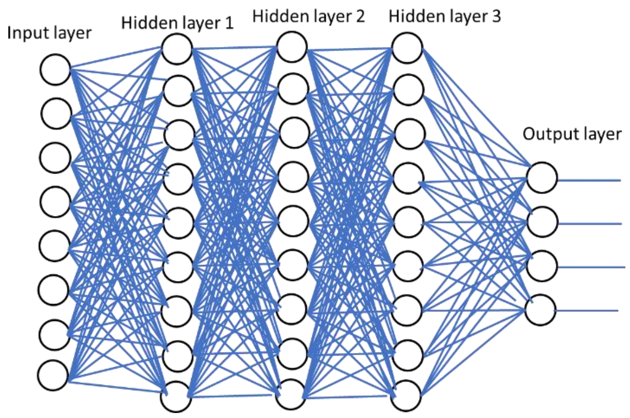


Fig.6.2 multiple layer neural network

Now let’s focus on a particular layer. Let *L* be the total number of layers (not including the input layer, input layer is called layer [0], just input data). Thus, layer *l*, *l=1,2,..,L-1*, is a hidden layer, and layer *L* is the output layer. In general, layer *l* can be defined by weight matrix $W^{[l]}$ and bias vector $b^{[l]}$, and an activation function $g^{[l]}(\,)$. The input and output of layer *l* are denoted by vectors $a^{[l-1]}$ and $a^{[l]}$, respectively. The block diagram representation of layer *l* is shown in Fig.6.3. The dimensions or shapes of $W^{[l]}$ and $b^{[l]}$ are determined by the number of units in layer *l* and layer *l-1* which are denoted by $n^{[l]}$ and $n^{[l-1]}$.

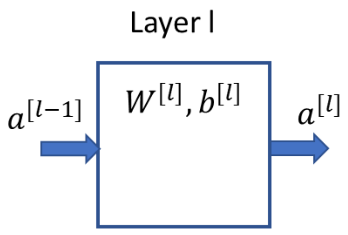


Fig. 6.3 diagram of layer *l*.

Suppose the numbers of units in layer *l* is $n^{[l]}$. Now let’s zoom in layer *l* with parameters $W^{[l]}$ and $b^{[l]}$. The shape of $W^{[l]}$ is $(n^{[l]}, n^{[l-1]})$, and shape of $b^{[l]}$ is $(n^{[l]}, 1)$. The *ith* row in $W^{[l]}$ and

$b^{[l]}$ is associated with the *ith* unit in this layer. The relationship between the input $a^{[l-1]}$ and the output $a^{[l]}$ of layer l can be described as

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \tag{6.1.a}$$

$$a^{[l]} = g^{[l]}\left(z^{[l]}\right) \tag{6.1.b}$$

It is helpful to check the matrix shapes on both sides of the above equations by the rule of matrix multiplication. Check shape consistence for (6.1.a): $\left(n^{[l]},1\right)$=$\left(n^{[l]},n^{[l-1]}\right) \times \left(n^{[l-1]},1\right) + \left(n^{[l-1]},1\right)$. Check shape consistence for (6.1.b): $\left(n^{[l]},1\right)$=$g^{[l]}\left(n^{[l]},1\right)$. Please note that (6.1.a) and (6.1.b) is based on one data example.

6.1.2 Forward propagation and backward propagation

Forward propagation and backward propagation are two important ingredients of neural network training process. The data flow, shown in Fig.6.4, from the input layer to the output layer, is referred as forward propagation (indicated by blue arrows), which involves computing of (6.1.a) and (6.1.b) for all layers. The purpose of calculating forward propagation is two-fold: 1) to deliver quantities (indicated by green arrows) which are needed for computing gradients during the training process; 2) to predict the output for an input *x* during the inference for new data examples. The backward propagation (indicated by red arrows) will provide all derivatives of cost function with respect to parameters, which are needed in the gradient descent algorithm. The gradient descent algorithm updates the parameters until the optimal parameters have been found according to a stopping criterion.

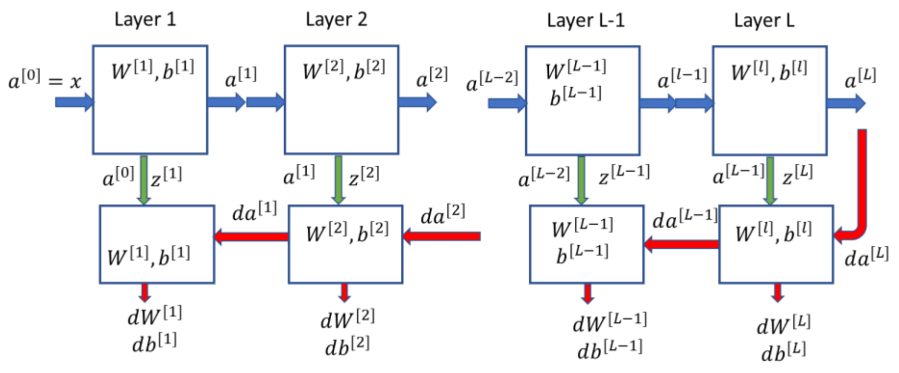


Fig.6.4 forward and backward block diagram

In Fig.6.4, the top row of boxes shows the forward propagation, i.e. the neural network, and the bottom row shows the backward propagation for derivative computation. The computations in the forward propagation and the backward propagation can be described below.

1) Forward propagation for layer l. $l = 1, 2, \ldots L$

   <u>For one data example *x*</u>

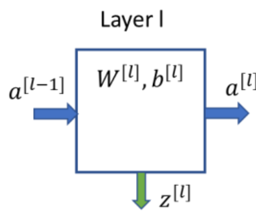
   $a^{[0]} = x$

   Input: $a^{[l-1]}$ (from cache, i.e., the output of the previous layer)

   Output: $a^{[l]}$, $z^{[l]}$

   $$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

   $$a^{[l]} = g^{[l]}\left(z^{[l]}\right)$$



(these outputs are saved to a cache for further use in layer *l*+1 computation and backward propagation)

<u>For *m* data examples (vectorized)</u> (shape)

$A^{[0]} = X$ (n_x, m)

Input: $A^{[l-1]}$ (from cache)

Output: $A^{[l]}$, $Z^{[l]}$(saved to cache)

$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ $(n^{[l]}, m)=(n^{[l]}, n^{[l-1]}) \times (n^{[l-1]}, m) + (n^{[l-1]}, 1)$

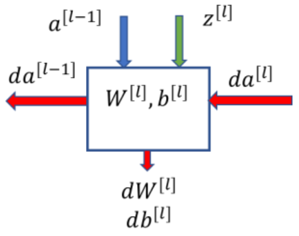
$A^{[l]} = g^{[l]}(Z^{[l]})$ $(n^{[l]}, m)$

(these outputs are saved to a cache for further use in layer *l*+1 computation and backward propagation, note that $+b^{[l]}$ is a broadcasting adder operation to different columns)

2) Backward propagation for layer *l*

Input: $da^{[l]}$(from layer *l*+1 backward propagation), $a^{[l-1]}, z^{[l]}$ (from cache)

Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$



<u>For one data example *x*</u> (shape)

Output layer ($l = L$)

$da^{[L]} = -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}$ (assume sigmoid function in the output layer)

$da^{[L]} = \frac{dL(\hat{y},y)}{da^{[L]}} = -\frac{y}{a^{[L]}}$ (assume softmax in the output layer)

$dz^{[L]} = a^{[L]} - y$ (for either sigmoid or softmax)

Hidden layers ($l = L-1, L-2, \dots, 2, 1.$)

$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$ $(n^{[l]}, 1) = (n^{[l]}, 1) * (n^{[l]}, 1)$

$dW^{[l]} = dz^{[l]}a^{[l-1]^T}$ $(n^{[l]}, n^{[l-1]}) = (n^{[l]}, 1) \times (n^{[l-1]}, 1)^T$

$db^{[l]} = dz^{[l]}$ $(n^{[l]}, 1) = (n^{[l]}, 1)$

$da^{[l-1]} = W^{[l]^T} dz^{[l]}$ $(n^{[l-1]}, 1) = (n^{[l]}, n^{[l-1]})^T \times (n^{[l]}, 1)$

<u>For *m* data examples (vectorized)</u>

Output layer ($l = L$)

$dA^{[L]} = -\frac{Y}{a^{[L]}} + \frac{1-Y}{1-a^{[L]}}$ $(1, m)$ for sigmoid function at layer L

$dA^{[L]} = -\frac{Y}{a^{[L]}}$ $(n^{[L]}, m)$ for softmax function at layer L

$dZ^{[L]} = A^{[L]} - Y$ $(n^{[L]}, m)$ for either sigmoid or softmax

Hidden layers ($l = L-1, L-2, \dots, 2, 1.$)

$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$ $(n^{[l]}, m) = (n^{[l]}, m) * (n^{[l]}, m)$

$dW^{[l]} = dZ^{[l]}A^{[l-1]^T} * (\frac{1}{m})$ $(n^{[l]}, n^{[l-1]}) = (n^{[l]}, m) \times (n^{[l-1]}, m)^T$

$db^{[l]} = \left(\frac{1}{m}\right) np.sum(dZ^{[l]}, axis = 1, keepdims = True)$ $(n^{[l]}, 1) = (n^{[l]}, 1)$

$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]} \qquad \left(n^{[l-1]}, 1\right) = \left(n^{[l]}, n^{[l-1]}\right)^T \times \left(n^{[l]}, 1\right)$$

Attention should be paid to the differences when different activation functions (sigmoid for binary classification or softmax for multiple classification) are applied in the output layer. In the case of softmax, Y is encoded as one-hot code matrix with a shape of $\left(n^{[L]}, m\right)$. Although $dA^{[L]}$ has different equations for sigmoid and softmax functions, $dZ^{[L]} = A^{[L]} - Y$ is the same for both. In the gradient descent algorithm, the parameters can be updated as

$$W^{[l]} \leftarrow W^{[l]} - \alpha \cdot dW^{[l]} \tag{6.2.a}$$

$$b^{[l]} \leftarrow b^{[l]} - \alpha \cdot db^{[l]} \tag{6.2.b}$$

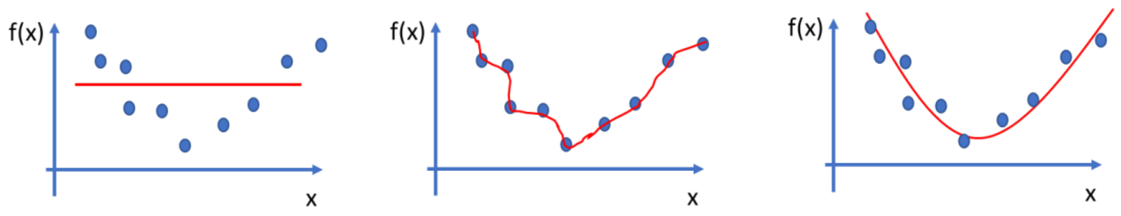
6.2 Generalization and Model Selection

6.2.1 Generalization, Underfitting and Overfitting

In a standard supervised learning setting, we train the model based on the training dataset, which is usually a small portion of the entire possible data space. The goal of learning the model is to discover the pattens of the entire data space. For example, in a task of classifying a picture as cat or dog, we first collect a training dataset, which includes many (e.g. hundreds or thousands of) pictures of either cat or dog. After trained by the training set, the model is expected to recognize any picture with either cat or dog from a test dataset, which the model has never seen before. This capability of the model is called *generalization*. This generalization capability assumes that both the training dataset and the test dataset are drawn independently from an identical distribution. This is commonly called the *i.i.d* (**i**dentical **i**ndependent **d**istribution) assumption.

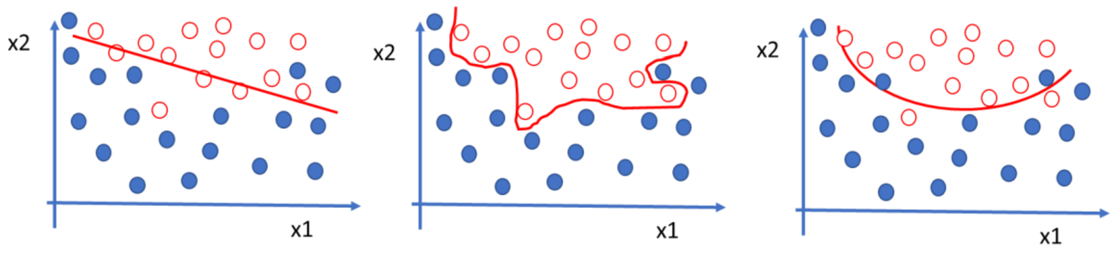
A model with low generalization capability usually demonstrates a phenomenon known as *overfitting*: the model fits the training data more closely than it fits the underlying distribution. There are two factors that affect the generalization of a model: 1) model complexity; and 2) training dataset size. There is no strict definition of model complexity. Intuitively, a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. With neural networks, a model that takes more training iterations is considered more complex. The training dataset size is measured by the number of training examples. In general, the more complex the model with a small training dataset, the more likely overfitting occurs. Opposite to overfitting, *underfitting* occurs when the model is too simple to capture the pattern of the training dataset.

Fig.6.5 illustrates three results of a curve fitting problem. Fig.6.5(a) shows the underfitting result, where the model is the linear line. Fig.6.5(b) shows the overfitting result. In Fig.6.5(b), although the error between the curve and the training examples is very small (perfectly fitting), if we generate some new data examples, then these examples might be far away from the curve. The overfitting model captures the noise pattern of the training set, that is not the true relationship between the input and the target. Fig.6.5(c) shows the appropriate fitting curve that is quadratic. Similarly, Fig. 6.6 shows the underfitting and overfitting for a classification task.



(a) underfitting (b) overfitting (c) appropriate fitting

Fig.6.5 Underfitting and overfitting in curve fitting



(a) underfitting (b) overfitting (c) appropriate

Fig.6.6 Underfitting and overfitting in classification

6.2.2 Training set, Validation set, and Test set

To develop a machine learning product, we usually divide the original dataset into three non-overlapping subsets: *training set*, *validation set*, and *test set*. In deep learning, the training set is used to learn the weights (or parameters) of candidate models. We use the validation set to test each candidate model, so that we can compare the models and select the best candidate model. The test set is reserved for testing the final selected model.

Ideally, all the data are expected to be drawn from the same underlying distribution (*i.i.d* assumption). It is a common practice to maintain an approximate same categorical ratio in the three subsets, which is called stratified sampling. The percentages of three parts are typically 60%, 20% and 20% for training, validation and test sets, respectively. However, if the number of data examples is very large (e.g. more than 1,000,000), the percentages for validation 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% validation), and 10,000 (1% test).

To diagnose the overfitting or the underfitting of a model, we can examine the training error and the validation error. The training error is the error when the trained model is tested on the training set, while the validation error is the error when the trained model is tested on the validation set. In a sense of statistics, there is no reason for a trained model to perform better on the validation set than on the training set because the model has never seen the validation data before. Thus, in general, the training error is always smaller than the validation error.

When the model underfits the training set, both the training error and the validation error are unacceptably large. This implies that the model is too simple to capture the basic pattern of the training set. For example, this will happen when we use a linear function to fit a dataset generated by a quadratic function. With the training set size fixed, we can alleviate the underfitting problem by increasing the complexity of the model. As the result, both the training error and the validation error are expected to decrease. However, if the complexity of the model is increased too much, the validation error will increase while the training error keeps decreasing. The model will lose the generalization capability. This indicates the occurrence of overfitting.

For examples, consider the following cases:

1) Training set error — 1% and validation set error — 10% means that our model is overfitting train set and not being able to generalize unseen examples.
2) Training set error — 10% and validation set error — 11% means that our model is underfitting the training set.
3) Training set error — 0.5% and validation set error — 1% means that our model is performing well, and we can test this model on the test set.

On the other hand, for an overfitting model, we can increase the training set size to alleviate or avoid the overfitting. Thus, whether underfitting or overfitting can depend on both the complexity of the model and the training set size.

Fig.6.7 illustrates the relationship between overfitting/underfitting and model complexity, given the fixed training set. By examining the error curves, we can tell whether the model is overfitting or underfitting. The general strategies to avoid underfitting or overfitting are shown in Fig.6.8. we will treat regularizations in Section 6.3.

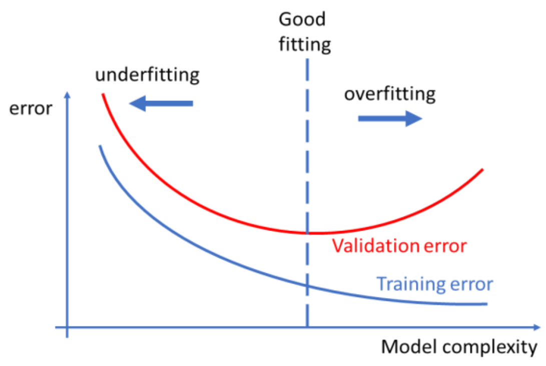


Fig. 6.7 Relationship between overfitting and model complexity

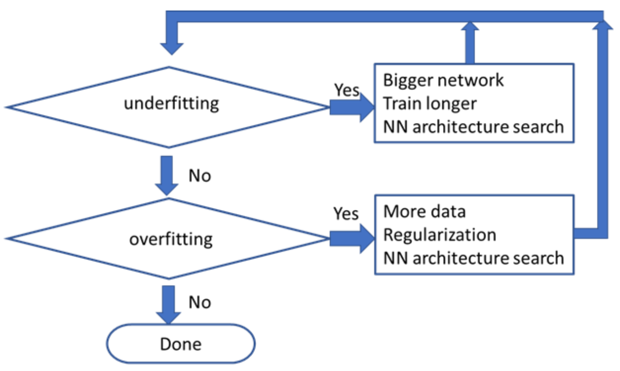


Fig.6.8 Strategies to avoid underfitting and overfitting

6.2.3 Model Selection and k-Fold Cross-Validation

In machine learning, we usually select our final model after evaluating serval candidate models on validation dataset. The final model should have the best generalization performance. This process is called *model selection*.

Model selection involves *hyperparameter selection* and model type selection. Hyperparameters control the effective complexity of the model. For example, when we want to fit a sequence of data using a polynomial curve, the order of the polynomial controls the number of free parameters in the model and thereby governs the model complexity. There is an optimal order that gives the best generalization. With neural networks, we may wish to compare models with different numbers of layers, different numbers of units in each layer, and different choices of activation functions. The parameters that control the training process, such as learning rate, the number of iterations, optimization method, and weight initialization method, also have impacts on the performance of the model. All these parameters that controls either the model complexity or training process are called hyperparameters. Furthermore, we may also wish to consider a range of different types (e.g. decision trees, support vector machines, or neural networks) of model in order to find the best one for a particular application.

In the previous section, we mentioned that a validation dataset is hold out for testing and comparing the generalization performance of the candidate models. In many applications, however, the availability of the original dataset is very limited, and we wish to use as much of the available data as possible for training. But, if the validation set is small, the performance estimate will be sensitive to the content of the validation set, and thereby the performance estimate has a high variance. One solution to this problem is to use *K-fold cross-validation*, which is illustrated in Fig.6.9 for K=5.

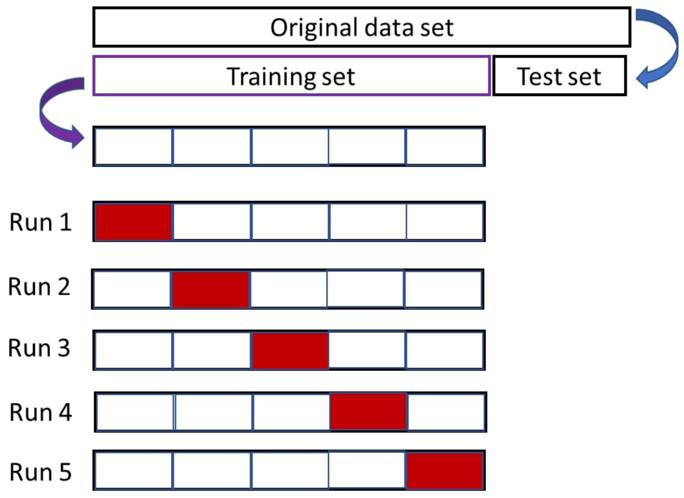


Fig.6.9 K-fold cross-validation (K=5): the red box indicates the validation set for the run.

To implement K-fold cross-validation, we split the original dataset into two parts: training set and test set. We randomly partition the training set into K groups, each of which has the same size. Then we perform K runs. For each run, say the *k-th* run, we evaluate the model using the k-th group while training the model using the remaining K-1 groups. After completing K runs, the training and validation errors can be obtained by averaging the results of the K runs. This allows (K-1)/K of the available data to be used for training while making use of all of the data to evaluate performance. This average of the K runs will be less noisy, and be used for model selection.

The typical value for K in K-fold cross-validation is 10. However, if the training set is relatively small, it may be helpful to increase K, which results in more data to be used for training in each run. The extreme case is K=N, where N is the number of data examples in the training set, which gives the *leave-one-out* technique. One major drawback of cross-validation is the computational cost because the higher the K value, the more training runs are needed. Thus, on the other hand, if the training set is relatively large, we can decrease the value of K (e.g. 4 or 5) to reduce the training time.

6.3 Regularization

As we discussed in the previous section, to alleviate the overfitting problem, we can reduce the model complexity or increase the training data. A model with high variance is likely overfitting train set and not being able to generalize unseen examples. A theoretical analysis of the bias and variance for a model can be found in Section 3.4. Since collecting more data is usually prohibitively expensive in practice, reducing the model complexity is often a practical solution to the overfitting issue.

Indeed, simply reducing the number of parameters of the model may avoid overfitting. For example, in the case of polynomial curve fitting task, we can use a low-order polynomial for the model. In the case of neural networks, we can use a neural network with a smaller number of layers and/or a smaller number of units in each layer. However, this may easily lead to underfitting.

In practice, an effective technique to control the model complexity is regularization. Weight regularization is the most widely used one for regularizing the model. In weight regularization, the norm of weight vector is a measure of the model complexity. To reduce the model complexity, we just need to decrease the norm of the weight vector at the cost of increased training error. We minimize the objective function, which is the sum of the loss and the norm of the weight vector. If the weights are too large, the algorithm might mainly focus on minimizing the weights. Another effective regularization technique for neural networks is dropout, which will be addressed in Section 6.3.4.

6.3.1 Regularization for Linear Regression

Consider a linear regression model $h_\theta(\mathrm{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$. The modified cost function for regularization is given by

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(e^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$$

$$= \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right] = \frac{1}{2m}\left[(\mathbf{X}\cdot\theta - Y)^T(\mathbf{X}\cdot\theta - Y) + \lambda \sum_{j=1}^{n} \theta_j^2\right] \quad (6.3)$$

where X is the matrix of training dataset with each row representing one example, Y is the column vector of labels of the training set, and θ is the column vector of weights. In the above equation, we regularize all the parameters by adding the penalty of the parameters' magnitudes to the square-error based cost function. Please note that θ_0 is not included in the penalty term. To learn the model, we minimize the modified cost function:

$$\min_{\theta} J(\theta) \tag{6.4}$$

The λ, or lambda, is the regularization parameter, a non-negative hyperparameter. It controls the degree of regularization. A larger λ will lead to a heavier regularization. Under the regularization, the hypothesis model will have a smooth output, hence with a reduced overfitting. If λ is too larger, it may smooth out the function too much and cause underfitting. On the other hand, if λ is too small or zero, the model is reduced to the case without regularization.

Now we can calculate the gradient of the regularization cost function. From Chapter 3 *Linear Regression*, we have the gradient of cost function without considering regularization

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \tag{6.5}$$

$$grad = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m} X^T \cdot (X \cdot \theta - Y) \tag{6.6}$$

Consider the term $\lambda \sum_{j=1}^{n} \theta_j^2$ added to the cost function for regularization (6.3), we can have the gradient with regularization as

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_0^{(i)} \tag{6.7a}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j \qquad j=1,2,\ ....,\ n \tag{6.7b}$$

Its vectorized form is

$$grad = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m} X^T \cdot (X \cdot \theta - Y) + \frac{\lambda}{m}\begin{bmatrix} 0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \tag{6.8}$$

6.3.2 Regularization for logistic regression

The cost function of logistic regression, without regularization, can be represented by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right)\ln\left(1 - \hat{y}^{(i)}\right)\right] \tag{6.9}$$

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

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|W\|_2^2 \tag{6.10}$$

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

The coefficient ½ and squaring L2 norm make the computation easy.

L1 regularization is given by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|W\|_1 \tag{6.11}$$

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

L2 regularization places a large penalty on large components of the weight vector. This biases our learning algorithm towards models that distribute weight evenly across a larger number of features. In practice, this might make them more robust to measurement error in a single variable. By contrast, L1 penalties lead to models that concentrate weights on a small set of features by clearing the other weights to zero. This is called feature selection, which may be desirable for some applications.

6.3.3 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,

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_F^2 \tag{6.12}$$

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

$$dW^{[l]} = dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) + \frac{\lambda}{m}W^{[l]} \tag{6.13}$$

The parameter update is given by

$$\begin{aligned} W^{[l]} := W^{[l]} - \alpha * dW^{[l]} &= W^{[l]} - \alpha * dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) - \alpha\frac{\lambda}{m}W^{[l]} \\ &= \left(1 - \alpha\frac{\lambda}{m}\right)W^{[l]} - \alpha * dZ^{[l]}A^{[l-1]^T} * \left(\frac{1}{m}\right) \end{aligned} \tag{6.14}$$

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 intuitively by examining the curve of an activation function, say tanh(z), in Fig.6.10. If λ is increased to impose a regularization, $W^{[l]}$ will be reduced. Then

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{6.15}$$

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.

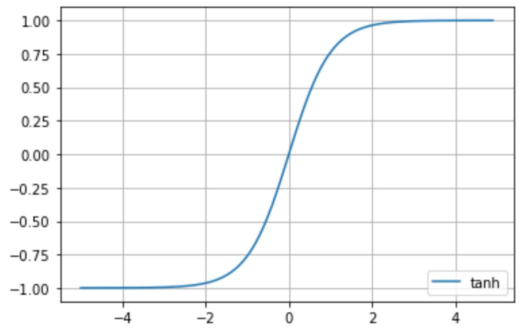


Fig.6.10 curve of tanh(z)

6.3.4 Dropout for regularization

Dropout is another technique to address the overfitting problem in neural networks. The key idea is to randomly drop (i.e., remove) 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.6.11. The choice of 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 by using a validation set or can simply be set at 0.5, which is an appropriate empirical value 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.

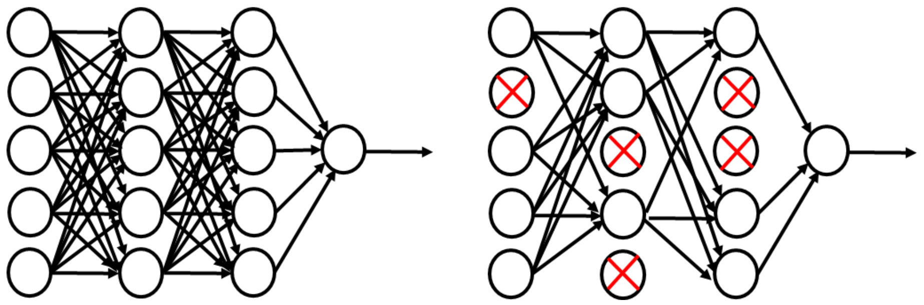


Fig.6.11 Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

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.

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 the output layer L) can be done in the same way.

Forward dropout:

```
A2 = relu(Z2)        # activation without dropout
D2 = np.random.rand(A2.shape[0], A1.shape[1])
D2 = (D2<keep_prob)      # mask for dropout, keep_prob is the retain prob.
A2 = np.multiply(A2,D2)  # activation with dropout
A2 = A2/keep_prob        # adjust the value
```

Note that D2 is a random binary matrix, and its shape is $(n^{[2]}, m)$, where $n^{[2]}$ is the number of units in layer 2, and m is the number of training examples. Thus, D2 has the same shape as A2, and serves as a mask to specify which units in A2 will be delivered to the next layer. Consider an example with 5 units in layer 2, and 4 training examples, i.e., $(n^{[2]}, m) = (5, 4)$. During a training iteration, D2 could be

$$D2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

During this training iteration, the above content of D2 shows that the model drops out unit 3 and unit 5 for the first data example (according to the first column in D2), and drops out unit 1 for the second example (indicated by column 2 in D2), and so on.

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

```
dA2 = np.dot(W3.T, dZ3)    # derivatives without dropout
### dropout
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. It is important to note that dropout is only applied at the training phase. In other words, no dropout will be applied for prediction at test time. A complete tutorial is provided in Section 6.9.

6.4 Weight Initialization

6.4.1 Random initialization

In gradient descent algorithms, the weights of the neural networks are updated from a set of initial values. The initial values of the weights play an important role in how quickly and to which local optimum the weights converge, because the loss function of a deep neural network is a complex, high-dimensional and non-convex function with many local minima. It is obviously not an option to initialize all weights to a zero or constant value. With the same initial weights for all nodes in a layer, the nodes are indistinguishable and redundant.

Random initialization is a widely used technique for the weight initialization. In random initialization, the initial values of the weights (W) are randomly generated from a distribution with mean zero and variance σ^2. In practice, the bias (b) in each node is usually initialized to zero. There are two type of distributions which are typically used for the initialization. One is the normal distribution, denoted as $\mathcal{N}(0, \sigma^2)$, while another is the uniform distribution, denoted as $U[-a, a]$. Note that the variance of the uniform distribution is $\sigma^2 = \frac{(2a)^2}{12} = \frac{a^2}{3}$.

The selection of the variance is critical for the effectiveness of training process. A large variance likely generates large magnitudes of the initial weights. If the variance is too small, the activations of the nodes with ReLU will vanish in the forward propagation or the nodes with sigmoid or tanh activation lose the non-linearity. If the variance is too large, the outputs of the nodes with ReLU will likely overflow due to a large value or the activation function (e.g. sigmoid and tanh) works in the saturation region, which leads to gradient vanishing.

6.4.2 Xavier initialization

To alleviate the problems of gradient vanishing or exploding, we initialize the weights in such way that the variances of the output and the input for a node are equal. Consider layer *l*, denoted as $a^{[l]}$. Its input is composed of the outputs of all nodes in the previous layer, denoted as $a^{[l-1]}$. Thus, the *i*-th node in layer *l* is

$$a_i^{[l]} = g\left(\sum_{j=1}^{n^{[l-1]}} w_{ij}^{[l]} a_j^{[l-1]}\right) \tag{6.16}$$

where g is the activation function, $w_{ij}^{[l]}$ is the weight from node *j* in layer *l*-1 to node *i* in layer *l*, $a_j^{[l-1]}$ is the output of node *j* in layer *l*-1, $n^{[l-1]}$ is the number of nodes in layer *l*-1, i.e., the fan-in of a node in layer *l*.

Assume that g is the identity function (i.e., $g(z) = z$) for simplicity, and the weights $w_{ij}^{[l]}$ are mutually independent and share the same distribution with zero mean, and the input $a_j^{[l-1]}$ are also independent and share the same distribution with zero mean, and the weights and the input are independent of each other. Then, the mean of the output $a_i^{[l]}$ is zero, and the variance is

$$Var[a_l] = n^{[l-1]}\, Var[w_l] \cdot Var[a_{l-1}] \tag{6.17}$$

where a_l, w_l, and a_{l-1} represent the random variables of each element in $a^{[l]}$, $W^{[l]}$, and $a^{[l-1]}$ respectively.

This implies that, passing through a node, the variance is scaled by a factor of $n^{[l-1]}\ Var[w_l]$. For the variances of the input and the output to be equal, we have

$$n^{[l-1]}\ Var[w_l] = n^{[l-1]}\sigma^2 = 1 \tag{6.18}$$

To account for both the forward propagation and backward propagation, we set the average of the forward and the backward scale factors to be a unit,

$$\frac{n^{[l-1]}+n^{[l]}}{2}\ \sigma^2 = 1 \tag{6.19}$$

Thus,

$$\sigma^2 = \frac{2}{n^{[l-1]}+n^{[l]}} \tag{6.20}$$

Derivation of (6.20) requires the assumption of the identity activation function. Since the initial weights and the inputs are relatively small, the activation function (e.g., sigmoid and tanh) typically works in its approximate linear region where its input is close to zero. Equation (6.20) is directly applicable for the tanh activation function because $\frac{d}{dz}\tanh(z)\,|_{z=0} = 1$.

For the nodes with a sigmoid activation function, since $\frac{d}{dz}\sigma(z)\,|_{z=0} = \frac{1}{4}$, the variance of the initial weights is adjusted accordingly,

$$\sigma^2 = 16 \times \frac{2}{n^{[l-1]}+n^{[l]}} \tag{6.21}$$

The initialization methods, defined by (6.20) and (6.21) for tanh and sigmoid activations respectively, are called Xavier initialization.

6.4.3 He initialization

ReLU is one of the most commonly used activation functions in deep learning (especially in convolutional neural networks), defined as

$$ReLU(z) = max(0, z) \tag{6.22}$$

Xavier initialization sometimes does not work well if the activation function is ReLU. He initialization was proposed to initialize the weights for layers with ReLU activations in deep neural networks.

For layer *l* in the forward pass, the linear combination (assuming the bias is zero) is

$$z^{[l]} = W^{[l]}a^{[l-1]} \tag{6.23}$$

where $z^{[l]} \in \mathbb{R}^{n^{[l]}}$, $W^{[l]} \in \mathbb{R}^{n^{[l]}\times n^{[l-1]}}$, $a^{[l]} \in \mathbb{R}^{n^{[l-1]}}$. Based on the assumptions of independence on $W^{[l]}$ and $a^{[l-1]}$, we have

$$Var[z_l] = n^{[l-1]}Var[w_l a_{l-1}] \tag{6.24}$$

where z_l, w_l, and a_{l-1} represent the random variables of each element in $z^{[l]}$, $W^{[l]}$, and $a^{[l-1]}$ respectively. Since the mean of w_l is zero, (6.24) can be represented as

$$Var[z_l] = n^{[l-1]}Var[w_l]\ E\left[a_{l-1}^2\right] \tag{6.25}$$

It is worth noticing that $Var[a_{l-1}] \neq E\left[a_{l-1}^2\right]$ because $E[a_{l-1}] \neq 0$. Since $a_{l-1} = Max(0, z_{l-1})$ and z_{l-1} has zero mean and a symmetrical distribution around zero, we have

$$E\left[a_{l-1}^2\right] = \frac{1}{2} Var[z_{l-1}] \tag{6.26}$$

This leads to

$$Var[z_l] = \frac{1}{2} n^{[l-1]} Var[w_l]\, Var[z_{l-1}] \tag{6.27}$$

To avoid reducing or magnifying the magnitudes of input signals exponentially through L layers, we set the scale factor to one, i.e.,

$$\frac{1}{2} n^{[l-1]} Var[w_l] = 1 \qquad \text{for } l = 2, 3, \dots, L \tag{6.28}$$

Or

$$\sigma^2 = Var[w_l] = \frac{2}{n^{[l-1]}} \qquad \text{for } l = 2, 3, \dots, L \tag{6.29}$$

For the first layer (l = 1), we should have $n^{[0]} Var[w_1] = 1$ because there is no ReLU applied on the input signal. But the factor 1/2 does not matter if it just exists on one layer. So we also adopt (6.28) and (6.29) in the first layer for simplicity.

For the backward propagation, a similar result can be obtained. It has been demonstrated that (6.29) is sufficient to avoid gradients from being reduced or magnified exponentially. A reader can refer to paper (He 2015) for details.

In summary, it is important to properly initialize the values of weights W for an effective and efficient training process. We randomly generate the initial values for W by following either the normal distribution or uniform distribution, with a mean of zero and a variance of σ^2. The variance depends on the type of activation function and the number of inputs to the node. The details are summarized in the following table.

Activation function	**Initialization method**	**Normal distribution $\mathcal{N}(0, \sigma^2)$**	**Uniform distribution $U[-a, a]$**
Sigmoid	Xavier initialization	$\sigma^2 = 16 \times \frac{2}{n^{[l-1]} + n^{[l]}}$	$a = 4 \times \sqrt{\frac{6}{n^{[l-1]} + n^{[l]}}}$
Tanh	Xavier initialization	$\sigma^2 = \frac{2}{n^{[l-1]} + n^{[l]}}$	$a = \sqrt{\frac{6}{n^{[l-1]} + n^{[l]}}}$
ReLU	He initialization	$\sigma^2 = \frac{2}{n^{[l-1]}}$	$a = \sqrt{\frac{6}{n^{[l-1]}}}$

Since ReLU is the most frequently used activation function for modern deep neural networks, we will adopt He initialization in our examples later, which works well practically for sigmoid or softmax activation.

6.5 Mini-batch Gradient Descent

6.5.1 Three types of gradient descent

As discussed in previous chapters, we use a gradient descent algorithm to search for the optimal or suboptimal values of model parameters. In the gradient descent algorithm, each update of the parameters can be based on the entire dataset, a single data example, or a batch of data examples. This leads to three types of gradient descent algorithms – batch, stochastic, and mini-batch, respectively.

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 rate.

In contrast, stochastic gradient descent, often abbreviated SGD, calculates the gradients and updates the model on each example in the training dataset.

Mini-batch gradient descent is another variation of the gradient descent algorithm that randomly splits the training dataset into small batches. The batches typically have the same size, possibly except the last batch. During the training process, the model parameters are updated once per a batch until all batches are applied (i.e., the epoch is completed). In deep learning, one epoch is typically not enough for the model to converge. To start a new epoch, we should create batches again in the same random way.

Thus, batch gradient descent and stochastic descent are the two extreme cases of mini-batch gradient descent. Mini-batch gradient descent with the batch size equal to the training set size is identical to the batch gradient descent. On the other hand, with the batch size of one, the mini-batch gradient descent is the same as the stochastic gradient descent.

To understand the concepts of mini-batch gradient descent and epoch, let's consider the following example. Suppose we use a dataset that has 200 samples, and we train a neural network through a mini-batch gradient descent with a batch size of 5 for 1000 epochs. For each epoch, the dataset then is randomly split into 40 batches with each having 5 samples. During the training iteration loop, the weights of the model are updated when each batch of 5 samples passes through. Thus, the model will be updated 40 times during one epoch. Furthermore, since the epoch number is 1,000, the model will be updated 40,000 times during the entire training process.

6.5.2 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 mini-batches given a particular batch size, say 1000. Thus, there are total 5000 mini-batches with the batch size 1000. The batches are denoted by $\left(X^{\{t\}}, Y^{\{t\}}, t = 1,2, \ldots, 5{,}000\right)$. The gradient descent algorithm is applied on 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)
            $Z^{[l]} = W^{[l]}X^{\{t\}} + b^{[l]}$
            $A^{[l]} = g^{[l]}(Z^{[l]})$
            ...
            $A^{[L]} = g^{[L]}(Z^{[L]})$
            Compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{t\ batch} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2\cdot 1000}\sum_{l}\|W^{[l]}\|_F^2$
        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.6.12.

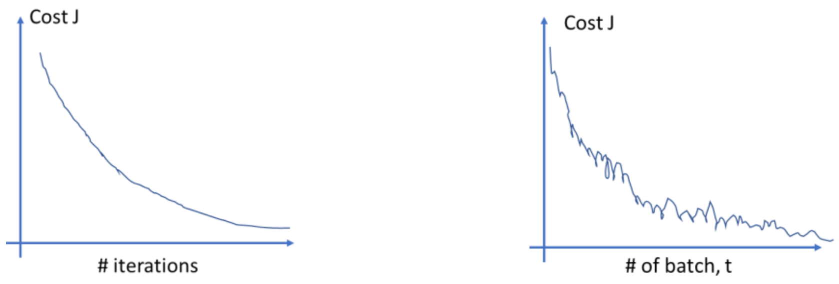


Fig.6.12 Cost curves. Left: batch gradient descent. Right: mini-gradient descent

6.5.3 Selection of mini-batch size

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

In general, a smaller value of batch size gives a learning process that updates the model more frequently at the cost of noise in the training process. Large values give a learning process that updates less frequently with accurate estimates of the error gradient. Fig.6.13 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 involves 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 hardware technology in 2022).

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.

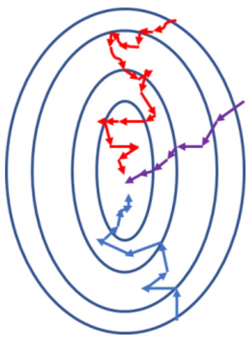


Fig.6.13 converge path

6.6 Normalization

In a machine learning pipeline, normalization or standardization of the training dataset is generally considered good practice, as it simplifies the input data distribution to a known standard distribution and thus accelerates the convergence of training process.

6.6.1 Input feature normalization

In many applications, the features in the data are measured by different scales. For example, the value of feature x_1 (e.g. square feet of a house) may vary in the range from 1 to 10000 while the range of feature x_2 (e.g. the number of bedrooms) varies from 1 to 5. 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:

$$z_i = \frac{x_i - u_i}{s_i} \tag{6.30a}$$

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

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i(j) \tag{6.30b}$$

$$s_i^2 = \frac{1}{m}\sum_{j=1}^{m} (x_i(j) - \mu_i)^2 \tag{6.30c}$$

6.6.2 Batch normalization

Training a deep neural network requires careful tuning of the model hyper-parameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters. It also suffers from a phenomenon, called *internal covariate shift*, which is defined as the change in the distribution of network activations due to the change in network parameters during training. The internal covariate shift presents a problem because the layers need to continuously adapt to the new distribution.

Batch normalization (BN) is an effective technique to address the issue of internal covariate shift. In addition, it allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularization, in some cases eliminating the need for dropout. Thus, batch normalization makes the training of deep neural networks faster and more stable. Batch normalization can be applied either right before or right after the activation function in each layer. For a fully connected neural network, each feature is normalized separately within a batch. Batch normalization for convolutional neural networks will be addressed in Chapter 8.

Fig.6.14 illustrates the bath normalization for layer *l*, which is applied right before the activation. For the *i*th data example in a batch, the linear combination is

$$X = WA^{[l-1]} + b \tag{6.31}$$

where $W \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$, $A^{[l-1]} \in \mathbb{R}^{n^{[l-1]}}$, $b \in \mathbb{R}^{n^{l}}$, $n^{[l]}$ and $n^{[l-1]}$ are the number of nodes in layer *l* and layer *l*-1 respectively. For a batch with a size of *m*, the variables $A^{[l-1]}$ and X will be extended by one more dimension with the size *m*. For example, we will use $X^{(j,i)}$ to denote the quantity at feature *j* (i.e. node *j*) and data example *i*. Fig.6.14 shows the case with $n^{[l-1]} = 5$, $n^{[l]} = 4$, batch size *m*=8. Each dash-line box represents a batch data for one feature.

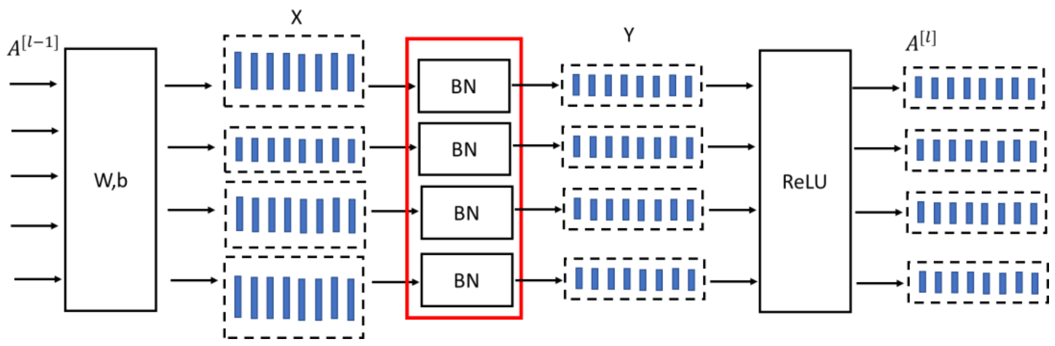


Fig.6.14 Batch normalization (red) applied in a neural network

Batch normalization is applied to each feature separately. The batch normalization for one feature, say feature *j*, can be described by the following equations

Let $x_i = X^{(j,i)}$

Mini-batch mean: $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$ (6.32a)

Mini-batch variance: $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$ (6.32b)

Normalize: $\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \qquad i = 1,2,\dots,m$ (6.32c)

Scale and shift: $y_i = \gamma\hat{x}_i + \beta, \qquad i = 1,2,\dots,m$ (6.32d)

y_i in (6.32d) is the result of batch normalization at feature *j* and example *i*. (6.32a) and (6.32b) calculate the mean and the variance within one batch, respectively. (6.32c) normalizes the batch data so that the input of each feature at each layer has a mean of zero and a variance of one. Note that ϵ is a small positive number to avoid dividing by zero. Two learnable parameters γ and β in (6.32d) are used to scale and shift the batch data, and thus to improve the representation ability of the network. The forward propagation in batch normalization, defined by (6.32), is illustrated in Fig.6.15, where L is the loss of the neural network.

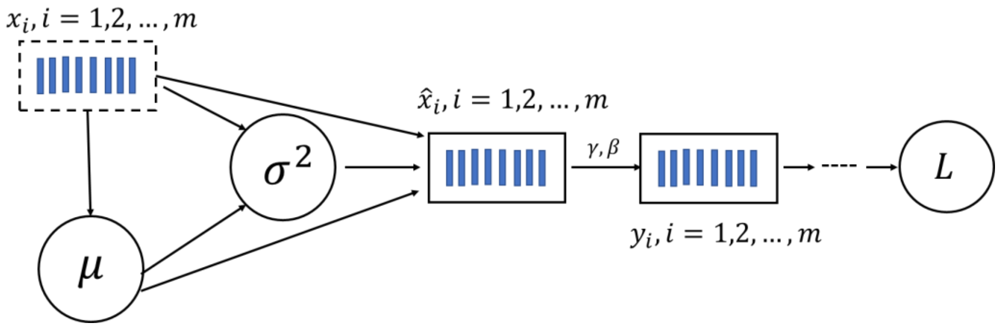


Fig.6.15 Dataflow in batch normalization

In backpropagation, we need to calculate the gradients of the loss with respect to the parameters and the input for each layer, given the gradient of the loss with respect to the output of this layer. specifically, consider the batch normalization in Fig.6.15, and suppose we know the results associated with the forward propagation (such as $x_i, \mu, \sigma^2, \hat{x}_i$), and $\frac{\partial L}{\partial y_i}, i = 1,2,\dots,m$. We will calculate $\frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta}, \frac{\partial L}{\partial x_i}, i = 1,2,\dots,m$.

By the calculus chain rule, we can easily have

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \tag{6.33}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\hat{x}_i \tag{6.34}$$

To calculate $\frac{\partial L}{\partial x_i}$, we compute the derivatives in a backward direction in Fig.6.15 as follows.

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i}\gamma \tag{6.35}$$

$$\frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}(x_i - \mu)(-\frac{1}{2})(\sigma^2 + \epsilon)^{-\frac{3}{2}} \tag{6.36}$$

$$\frac{\partial L}{\partial \mu} = \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu} + \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial \mu} = \frac{\partial L}{\partial \sigma^2}\frac{-2}{m}\sum_{i=1}^{m}(x_i - \mu) + \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}\frac{-1}{\sqrt{\sigma^2+\epsilon}} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}\frac{-1}{\sqrt{\sigma^2+\epsilon}} \tag{6.37}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i}\frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_i} + \frac{\partial L}{\partial \mu}\frac{1}{m} = \frac{\partial L}{\partial \hat{x}_i}\frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{\partial L}{\partial \sigma^2}\frac{2}{m}(x_i - \mu) + \frac{\partial L}{\partial \mu}\frac{1}{m} \tag{6.38}$$

During the training process, the computations (6.32-6.38) are based on a batch of data examples. However, during inference, we only have one data point, and thus need to estimate the global mean and the variance of x, using the mini-batch means and variances generated during the training process.

$$E[x] \leftarrow E_{\mathcal{B}}[\mu] \tag{6.39}$$

$$Var[x] \leftarrow \frac{m}{m-1}E_{\mathcal{B}}[\sigma^2] \tag{6.40}$$

where μ, σ^2 are the mean and the variance for a batch during the training process. In practice, to improve the computation efficiency, we can use the moving average to estimate the expectations in (6.39) and (6.40). For instance, during the training process, we can accumulate the batch means by moving average

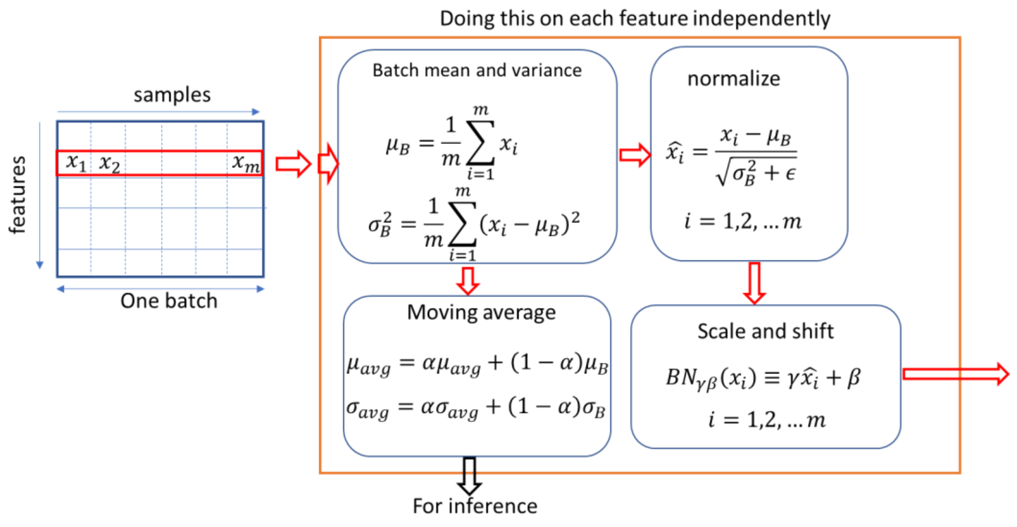
$$mean_{running} \leftarrow momentum \times mean_{running} + (1 - momentum) \times \mu \tag{6.41}$$

where the hyperparameter $momentum$, is a given coefficient between 0 and 1 At the end of training, $mean_{running}$ is an estimate of $E[x]$, and thus can be used in inference. $E_B[\sigma^2]$ in (6.40) can be similarly estimated by moving average.

In inference, the computation of the batch normalization is defined as

$$y_i = \gamma \frac{x_i - E[x]}{\sqrt{Var[x] + \epsilon}} + \beta \tag{6.42}$$

where γ, β are learned parameters. Thus, a batch normalization can be illustrated in Fig.6.16.



Doing this on each feature independently
samples
features
x_1 x_2 x_m
One batch
Batch mean and variance
$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$
$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
normalize
$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
$i = 1,2,\ldots m$
Moving average
$\mu_{avg} = \alpha\mu_{avg} + (1-\alpha)\mu_B$
$\sigma_{avg} = \alpha\sigma_{avg} + (1-\alpha)\sigma_B$
Scale and shift
$BN_{\gamma\beta}(x_i) \equiv \gamma\hat{x}_i + \beta$
$i = 1,2,\ldots m$
For inference

Fig.6.16 Batch normalization

6.7 Adam Optimization

6.7.1 Gradient descent with momentum

Due to the limited data amount in one batch when the batch size is small, there is significant oscillating in the converge path if the parameters are updated only based on the current batch gradients. In many applications, the gradient descent usually demonstrates a "*zigzag*" convergence path, as shown in Fig.6.17. Fortunately, the overall direction of the converge path eventually points to the minimal cost. 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 proceed, the recent gradients should have larger weights when the averaging is performed. Thus, we compute exponentially weighted averages over the gradients of all previous mini-batches. In fact, we already applied this technique to estimate the average in (6.41).

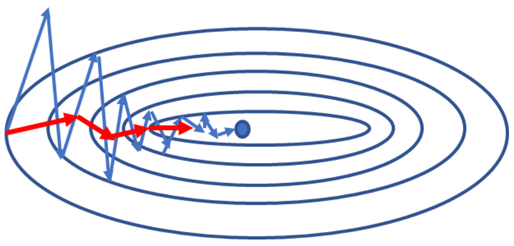


Fig.6.17 Gradient descent with momentum. The red path is obtained by averaging a few previous mini-batch gradients.

Consider an example of computing a smooth curve to approximate a series of noisy data samples $\{\theta_1, \theta_2, \dots \theta_t, \dots\}$. The exponentially weighted averages, also called moving averages, denoted by $\{v_1, v_2, \dots v_t, \dots\}$, can be used for this purpose, and computed by

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t \tag{6.43}$$

where β is a parameter for weight control $(0 < \beta < 1)$. Given zero initial value for v_t, (6.43) can be expressed by $\{\theta_1, \theta_2, \dots \theta_t, \dots\}$ explicitly

$$v_t = (1-\beta)(\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \cdots + \beta^{t-1}\theta_1) \tag{6.44}$$

(6.44) 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 (6.43) and (6.44) is a bias existing in the beginning of the process due to the zero initialization. A **bias correction** is available to compensate this initial bias

$$v_t \coloneqq \frac{v_t}{1-\beta^t} \tag{6.45}$$

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

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

```
$\mathrm{v}_\theta = 0$
Repeat {
    Get next $\theta_t$
    $\mathrm{v}_\theta \coloneqq \beta \mathrm{v}_\theta + (1-\beta)\theta_t$        (6.46)
}
```

Now let’s return to gradient descent with **momentum**. On each iteration *t* (mini-batch), 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*
    $v_{dw} = \beta v_{dw} + (1-\beta)dW$
```

$$v_{db} = \beta v_{db} + (1-\beta)db$$
$$W := W - \alpha v_{dw}$$
$$b := b - \alpha v_{db} \tag{6.47}$$

where dW and db are the derivatives of the loss function with respect to the weights and bias on the current batch. Note that there are two hyperparameters α, β for this update.

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

```
Initialization: $s_{dw} = 0, s_{db} = 0,$ $\varepsilon$: 10E-8 (avoid dividing by zero)

On iteration t:
    Compute dW, db on the current mini-batch
    $s_{dw} = \beta s_{dw} + (1-\beta)(\|dW\|^2)$
    $s_{db} = \beta s_{db} + (1-\beta)(\|db\|^2)$
    $W := W - \alpha \frac{dW}{\sqrt{s_{dw}}+\varepsilon},\ b := b - \alpha \frac{db}{\sqrt{s_{db}}+\varepsilon}$        (6.48)
```

6.7.2 Adam optimization algorithm

Adam optimization algorithm is a combination of gradient descent with momentum and RMS prop. 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
    $v_{dw} = \beta_1 v_{dw} + (1-\beta_1)dw,$
    $v_{db} = \beta_1 v_{db} + (1-\beta_1)db$
    $s_{dw} = \beta_2 s_{dw} + (1-\beta_2)(dw * dw),$
    $s_{db} = \beta_2 s_{db} + (1-\beta_2)(db * db)$
    $v_{dw} := \frac{v_{dw}}{1-\beta_1^t}$
    $v_{db} := \frac{v_{db}}{1-\beta_1^t}$
    $s_{dw} := \frac{s_{dw}}{1-\beta_2^t}$
    $s_{db} := \frac{s_{db}}{1-\beta_2^t}$
    $W := W - \alpha \frac{v_{dw}}{\sqrt{s_{dw}}+\varepsilon},\ b := b - \alpha \frac{v_{db}}{\sqrt{s_{db}}+\varepsilon}$        (6.49)
```

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.

6.7.3 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 stages, 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 one epoch as one pass through the dataset, as shown in Fig.6.18.

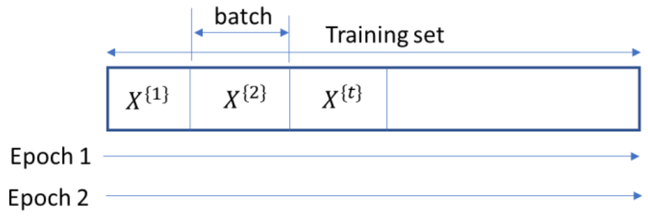


Fig.6.18 Epoch concept

An empirical learning rate decay is given by

$$\alpha = \frac{\alpha_0}{1+decay\_rate*epoch\_num} \tag{6.50}$$

Other empirical equations for learning rate decay include

$$\alpha = 0.95^{epoch\_num} \cdot \alpha_0 \tag{6.51}$$

$$\alpha = \frac{k}{\sqrt{epoch\_num}}\alpha_0 \tag{6.52}$$

where $epoch\_num$ is the index of epoch.

6.8 Gradient checking

The gradient computation is usually difficult to debug. We describe a numerical method to verify whether 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 not 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 $\frac{d}{d\theta}J(\theta)$.

$$g(\theta) = \frac{d}{d\theta}J(\theta) = \lim_{\varepsilon \to 0} \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \tag{6.53}$$

How can we check if our implementation of g is correct? In practice, we set ε to a small constant, say around 10^{-4}. Thus, given a function g(θ) that is supposed to compute $\frac{d}{d\theta}J(\theta)$ in the backpropagation, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \tag{6.54}$$

where $g(\theta)$ is the derivative we calculated in the backpropagation during the training process. The right-hand side of (6.54) shows the way we calculate the approximate derivative for the verification purpose.

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 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.

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 $\theta^{(i+)} = \theta + \varepsilon \times \vec{e}_i$, where

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{6.55}$$

where $\vec{e}_i$ is the i-th basis vector (a vector of the same dimension as θ, with a "1" for the i-th element and "0" for the rest elements). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ε. Similarly, let $\theta^{(i-)} = \theta - \varepsilon \times \vec{e_i}$, 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:

$$g_i(\theta) \approx g_{approx,i}(\theta) = \frac{J(\theta^{(i+)})-J(\theta^{(i-)})}{2\varepsilon} \tag{6.56}$$

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.*

$$d = \frac{\|g_{approx}(\theta)-g(\theta)\|_2}{\|g_{approx}(\theta)\|_2+\|g(\theta)\|_2} \tag{6.57}$$

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, because $g_{approx,i}(\theta)$ 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.
- 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)

6.9 Examples in Python

In this section, we will present how the practical considerations addressed earlier are implemented from scratch in Python. A reader is encouraged to read carefully and run the provided codes.

6.9.1 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.6.19. 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. We can specify a different number of units for each hidden layer by changing the corresponding number in the vector layers_dims. For example, we can change 20 to 10 for 10 units in the first hidden layer. However, if we want to build a network with more layers, we need to extend the Python code by explicitly adding more layers. Thus, this network can perform a binary classification. We will use this network to classify the dataset, which is plotted in Fig.6.20.

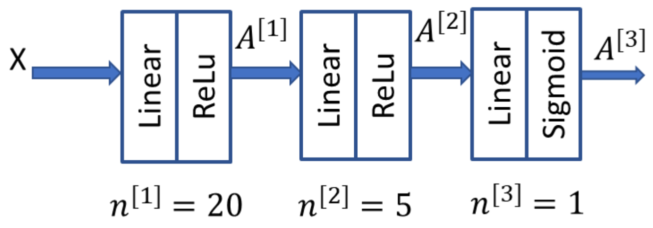
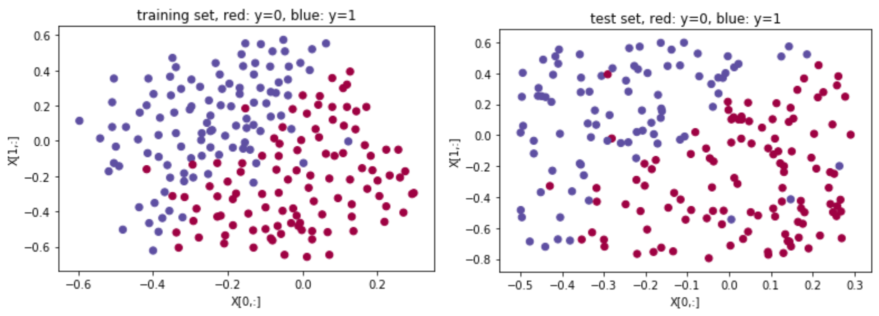


Fig.6.19 The implemented 3-layer network



(a) training set (b) test set

Fig.6.20 A dataset (data.mat) to be classified

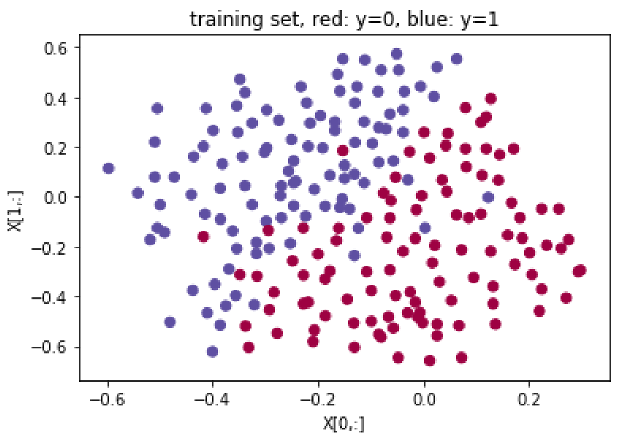
We implement this project in Jupyter notebook (regularization_dropout.ipynb) through the following steps:

1) Prepare the data. The training and test datasets are saved in a single file *data.mat*.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
```

```
def load_2D_dataset():
    data = scipy.io.loadmat('data.mat')
    train_X = data['X'].T
    train_Y = data['y'].T
    test_X = data['Xval'].T
    test_Y = data['yval'].T

    return train_X, train_Y, test_X, test_Y
```

```
# load and visualize dataset
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("training set, red: y=0, blue: y=1")
plt.show()
```



#train_X.shape: (2,211), train_Y.shape: (1,211)

#test_X.shape: (2,200), test_Y.shape: (1,200)

2) Define functions:

Activation functions: relu(x), sigmoid(x)

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

    return s
```

```
def sigmoid(x):
    """
    Arguments:
```

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

Forward propagation functions: 1) without dropout; and 2) with dropout

```
def forward_propagation(X, parameters):
    """
    Implements the forward propagation, without dropout

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

    Returns:
    A3 -- the output
    cache -- internal variables and weights
    """

    # 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 forward_propagation_with_dropout(X, parameters, keep_prob):
    """
    Implements the forward propagation with dropout
    keep_prob = 1 for no dropout

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    parameters -- python dictionary containing:
                    W1 -- weight matrix
                    b1 -- bias vector
                    W2 -- weight matrix
                    b2 -- bias vector
                    W3 -- weight matrix
                    b3 -- bias vector
    keep_prob - probability of keeping a neuron active during dropout, scalar
    Returns:
    A3 -- 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)
    ### randomly drop out in layer 1 ###
    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1<keep_prob)     ### dropout mask
    A1 = np.multiply(A1,D1)
    A1 = A1/keep_prob       ### scale large
    ## end code


    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
     ### drop out A2  ###
    D2 = np.random.rand(A2.shape[0], A1.shape[1])
    D2 = (D2<keep_prob)
    A2 = np.multiply(A2,D2)
    A2 = A2/keep_prob
    ## 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
```

Cost functions: 1) without regularization and 2) with regularization

```
def compute_cost(a3, Y):
    """
    Implement the cost function

    Arguments:
    a3 -- activation, output of forward propagation
    Y -- "true" labels vector, same shape as a3

    Returns:
    cost - value of the cost function
    """
    m = Y.shape[1]

    logprobs = np.multiply(-np.log(a3),Y) + np.multiply(-np.log(1 - a3), 1 - Y)
    cost = 1./m * np.nansum(logprobs)

    return cost
```

```python
def compute_cost_with_regularization(A3, Y, parameters, lambd):
    """
    adding L2 norm of weights to cross entropy cost
    """
    m = Y.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]

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


    # regularization part
    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
```

Backward propagation functions: 1) no regularization and no dropout; 2) with regularization; 3) with dropout.

```python
def backward_propagation(X, Y, cache):
    """
    Implement backward propagation without dropout and without regularization.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector (1 for yes, 0 for no)
    cache - cache 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
```

```python
def backward_propagation_with_regularization(X, Y, cache, lambd):
    """
    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    cache -- 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

    ### Regularization for W3 ###
    dW3 = 1./m * (np.dot(dZ3, A2.T) + lambd * W3)
    ######
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    ### regularization for W2 ###
    dW2 = 1./m * (np.dot(dZ2, A1.T) + lambd * W2 )
    ######
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    ### regularization for W1 ###
    dW1 = 1./m * (np.dot(dZ1, X.T) + lambd * W1 )
    ######
    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
```

```python
def backward_propagation_with_dropout(X, Y, cache, keep_prob):
    """
    Implement the backward propagation with dropout, but no regularization

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector of shape (output size, number of examples)
    cache -- from forward_propagation_with_dropout()
    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 for dropout ####
    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 for dropout
    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
```

Update parameter by gradient descent

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

    Arguments:
    parameters -- python dictionary containing the parameters: wi, bi
    grads -- python dictionary containing the gradients for each parameters:
             dWi, dbi
    learning_rate -- the learning rate, scalar.

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    n = len(parameters) // 2 # number of layers in the neural networks

    # Update rule for each parameter
    for k in range(n):
        parameters["W" + str(k+1)] = parameters["W" + str(k+1)] - learning_rate
* grads["dW" + str(k+1)]
        parameters["b" + str(k+1)] = parameters["b" + str(k+1)] - learning_rate
* grads["db" + str(k+1)]

    return parameters
```

Initialize parameters

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

    Returns:  He initialization
    parameters -- python dictionary containing parameters "W1", "b1", ...,
"WL", "bL":
        Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
        bl -- bias vector of shape (layer_dims[l], 1)

    """

    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(0.5*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
```

3) Model function: put all together, define the size of the network, initialize the parameters, define the training iteration loop including forward propagation, backward propagation, parameter updating, optionally computing and display cost. The model function returns the learned parameters.

```
def model(X,Y, learning_rate=0.3, num_iterations=30000, print_cost= True,
lambd=0, keep_prob=1):
    """
    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector
    learning_rate -- learning rate hyperparameter
    num_interations -- number of updates
    print_cost -- whether print cost
    lambd -- regularization parameter
    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
    """
    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_prob)
        #cost
        if lambd == 0:
            cost = compute_cost(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 thousands)')
    plt.title("learning rate = " + str(learning_rate))
    plt.show()

    return parameters
```

4) Prediction functions: accuracy, plot decision boundary

```
def predict(X, y, parameters):
    """
    This function is used to predict the results of an n-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    p = np.zeros((1,m), dtype = np.int)

    # Forward propagation
    a3, caches = forward_propagation(X, parameters)

    # convert probas to 0/1 predictions
    for i in range(0, a3.shape[1]):
        if a3[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    # print results

    #print ("predictions: " + str(p[0,:]))
    #print ("true labels: " + str(y[0,:]))
    print("Accuracy: "  + str(np.mean((p[0,:] == y[0,:]))))

    return p
```

```
def predict_dec(parameters, X):
    """
    Used for plotting decision boundary.

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (m, K)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """

    # Predict using forward propagation and a classification threshold of 0.5
    a3, cache = forward_propagation(X, parameters)
    predictions = (a3>0.5)
    return predictions
```

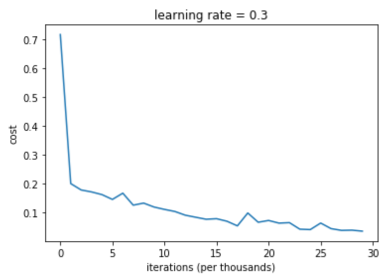
```
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),np.arange(y_min,y_max,h))
    # Predict the function value for the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y.ravel(), cmap=plt.cm.Spectral)
    plt.show()
```

5) Run the model and test

Without regularization and dropout. Overfitting occurs because training accuracy =0.95 and test accuracy= 0.92.

```
parameters = model(train_X, train_Y, lambd=0, keep_prob = 1.0)
```

```
cost after iteration 0: 0.7174841115578868
cost after iteration 1000: 0.20018269019856624
…
```



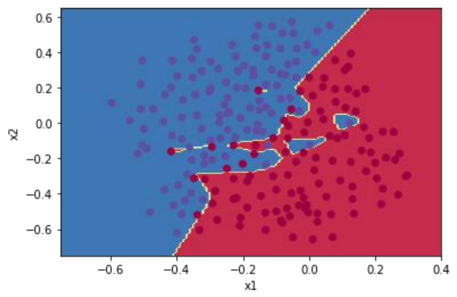
```
print("on the train set:")
predictions_train = predict(train_X, train_Y, parameters)
```

```
on the train set:
Accuracy: 0.95260663507109
```

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

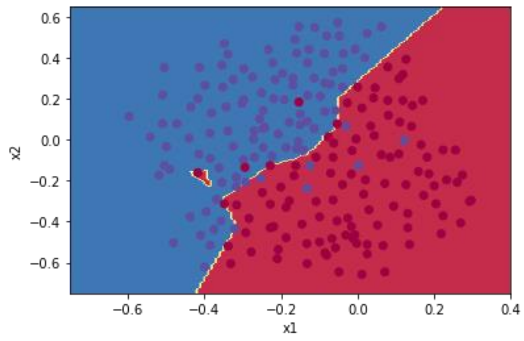
```
on the test set:
Accuracy: 0.92
```

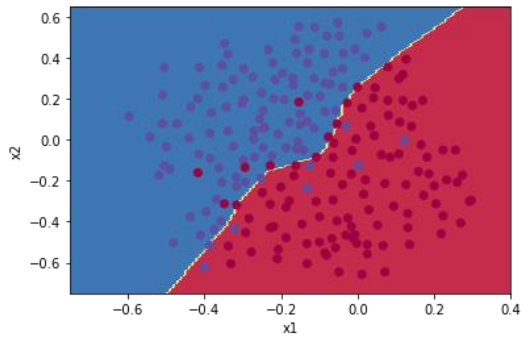
```
plt.xlim([-0.75, 0.40])
plt.ylim([-0.75, 0.65])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T),train_X,train_Y)
```



With regularization or dropout.

The statement `parameters = model(train_X, train_Y, lambd=0.1, keep_prob = 1)` implements regularization with λ=0.1, shown in Fig.6.21(a). Accuracy on training set is 0.95260663507109 and 0.95 on test set. The statement `parameters = model(train_X, train_Y, lambd=0, keep_prob = 0.8)` implements a dropout with keep probability of 0.8, shown in Fig.6.21(b). Accuracy is 0.9241706161137441 on training set and 0.935 on test set.





a) Weight regularization with λ=0.1 b) dropout with keep probability=0.8

Fig.6.21 Regularization a) weight regularization; b) dropout

6.9.2 A 3-layer network for multi-classification with mini-batch training and different optimization options

In this project, we will build a generic 3-layer neural network, as shown in Fig.6.22, where the activation function in the output layer uses softmax for multiple-class classification. The size of each layer is defined by the corresponding element in the list *layers_dims*. For example, *layers_dims* =[12288, 25,12,6] specifies 12288 features of input, 25 units in the first hidden layer and 12 units in the second hidden layer, 6 units in the output layer. All hidden layers use ReLU activation function while the output layer uses softmax for classification. Note that binary classification can be implemented by assigning 2 to *layers_dims*[3] with one-hot-coded labels. The purpose of this project is to demonstrate how mini-batch training, optimization method, and softmax are implemented from scratch. At the end, we will apply the designed neural network to two different tasks: a simple binary classification task and a 6-class image classification task.

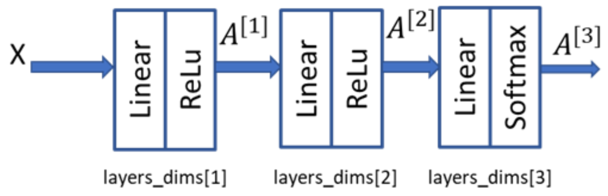


Fig.6.22 A generic 3-layer neural network

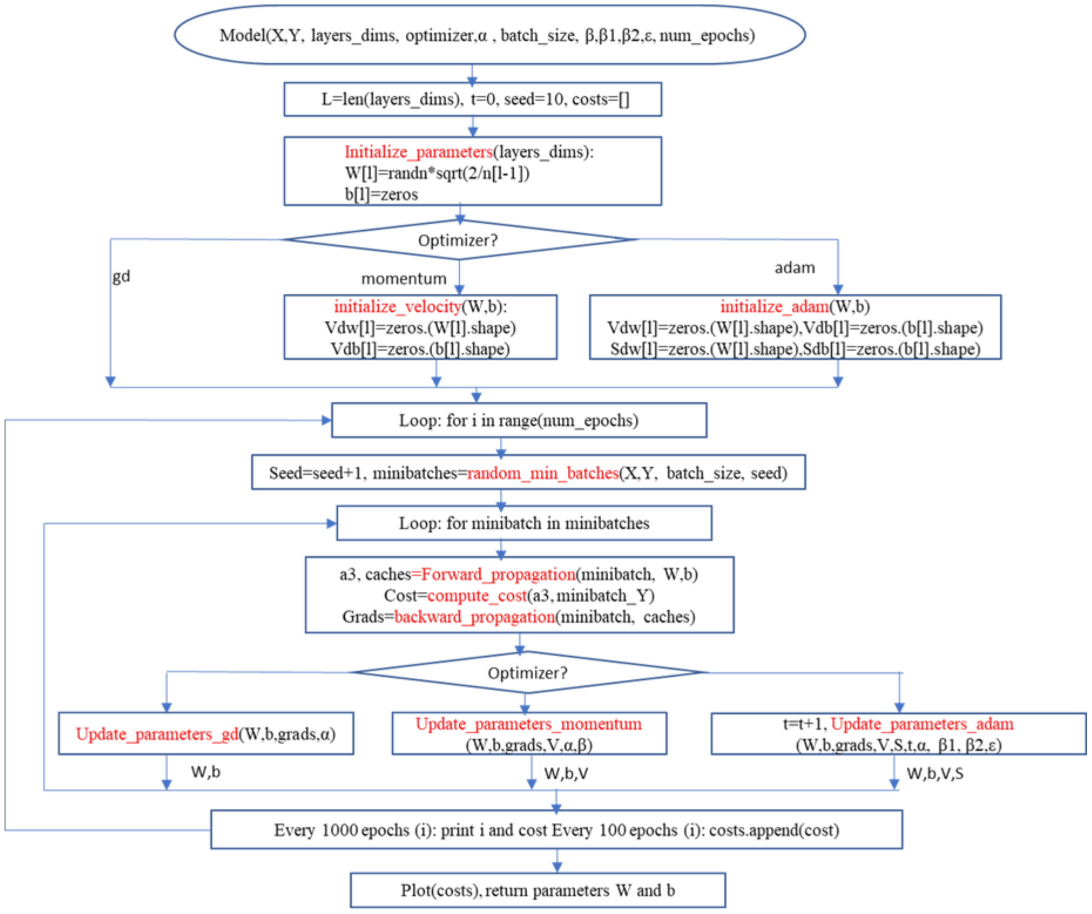
The design flowchart of the model is shown in Fig.6.23. The red fonts indicate functions defined separately. This flowchart can help a reader understand the overall picture of mini-batch concept and its implementation. It is helpful to refer to this flowchart when reading through the subsequent texts. We will train the neural network using mini-batch strategy to feed data, and using three different optimization options: basic gradient descent, momentum and Adam. The resulting network model in python is given by

```
model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size = 64,
      beta = 0.9, beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000,
      print_cost = True)
```

The values for arguments are default. The meaning of each argument will be explained in the following texts.

The model has the following arguments:
X -- input data, of shape (number of features, number of examples)
Y – one-hot-label, of shape (K, number of examples), where K is the number of classes.
layers_dims -- python list, containing the size of each layer
learning_rate α -- the learning rate, scalar.
mini_batch_size -- the size of a mini batch
beta -- Momentum hyperparameter
beta1 -- Exponential decay hyperparameter for the past gradients estimates
beta2 -- Exponential decay hyperparameter for the past squared gradients estimates
epsilon -- hyperparameter preventing division by zero in Adam updates
num_epochs -- number of epochs
print_cost -- True to print the cost every 100 epochs
Returns: parameters -- python dictionary containing your updated parameters

The parameters W and b are updated in two nested loops. The outer loop is designed for different epochs while the inner loop works for different batches within an epoch. Each epoch is obtained by partitioning the randomly shuffled dataset into mini-batches. Each update of parameters is based on one mini-batch. For example, if the total number of epochs is 10,000, and 5 mini-batches in each epoch, then the cost and the parameters W and b will be updated 50,000 times. Please note that we can choose to save and plot some costs (not all) which includes enough information for monitoring the behavior of training. When the mini-batch size is equal to one, the model implements stochastic gradient descent. When the mini-batch size is *m*, the model implements batch gradient descent. There are two levels of initializations: 1) basically initialize W and b; and 2) extra initialize V for momentum and (V,S) for Adam.



Model(X,Y, layers_dims, optimizer,α , batch_size, β,β1,β2,ε, num_epochs)
L=len(layers_dims), t=0, seed=10, costs=[]
Initialize_parameters(layers_dims):
W[l]=randn*sqrt(2/n[l-1])
b[l]=zeros
Optimizer?
gd
momentum
adam
initialize_velocity(W,b):
Vdw[l]=zeros.(W[l].shape)
Vdb[l]=zeros.(b[l].shape)
initialize_adam(W,b)
Vdw[l]=zeros.(W[l].shape),Vdb[l]=zeros.(b[l].shape)
Sdw[l]=zeros.(W[l].shape),Sdb[l]=zeros.(b[l].shape)
Loop: for i in range(num_epochs)
Seed=seed+1, minibatches=random_min_batches(X,Y, batch_size, seed)
Loop: for minibatch in minibatches
a3, caches=Forward_propagation(minibatch, W,b)
Cost=compute_cost(a3, minibatch_Y)
Grads=backward_propagation(minibatch, caches)
Optimizer?
Update_parameters_gd(W,b,grads,α)
Update_parameters_momentum
(W,b,grads,V,α,β)
t=t+1, Update_parameters_adam
(W,b,grads,V,S,t,α, β1, β2,ε)
W,b
W,b,V
W,b,V,S
Every 1000 epochs (i): print i and cost Every 100 epochs (i): costs.append(cost)
Plot(costs), return parameters W and b

Fig.6.23 Design flowchart of the neural network model

Now let’s go through the details of Python program (mini_batch.ipynb):

1) Basic activation functions: sigmoid(x), softmax(x), relu(x).

```
import numpy as np
import matplotlib.pyplot as plt
import math
import sklearn.datasets
```

```
def sigmoid(x):
    """
    """
    s = 1/(1+np.exp(-x))
    return s
def softmax(x):
    """
    Arguments:
```

```
    x -- A numpy array of any size.
    Return:
    s -- softmax(x)
    """
    t=np.exp(x)
    s = t/np.sum(t, axis=0)

    return s
def relu(x):
    """
    """
    s = np.maximum(0,x)

    return s
```

2) Update parameters using basic gradient descent

```
def update_parameters_with_gd(parameters, grads, learning_rate):
    """
    Update parameters using one step of gradient descent

    Arguments:
    parameters -- python dictionary containing parameters to be updated:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing gradients to update each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    learning_rate -- the learning rate, scalar.

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    L = len(parameters) // 2 # number of layers in the neural networks

    # Update rule for each parameter
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
            learning_rate*grads['dW' + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
            learning_rate*grads['db' + str(l+1)]

    return parameters
```

3) Update parameters using momentum gradient descent

```
def initialize_velocity(parameters):
    """
    Initializes the velocity as a python dictionary with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the
                  corresponding gradients/parameters.
    Arguments:
    parameters -- python dictionary containing parameters.
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl

    Returns:
    v -- python dictionary containing the current velocity.
                    v['dW' + str(l)] = velocity of dWl
                    v['db' + str(l)] = velocity of dbl
    """

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}

    # Initialize velocity
    for l in range(L):
        v["dW" + str(l+1)] = np.zeros(parameters['W'+str(l+1)].shape)
        v["db" + str(l+1)] = np.zeros(parameters['b'+str(l+1)].shape)

    return v
```

```
def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
    """
    Update parameters using Momentum

    Arguments:
    parameters -- python dictionary containing parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing gradients for each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- python dictionary containing the current velocity:
                    v['dW' + str(l)] = ...
                    v['db' + str(l)] = ...
    beta -- the momentum hyperparameter, scalar
    learning_rate -- the learning rate, scalar

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- python dictionary containing your updated velocities
    """

    L = len(parameters) // 2 # number of layers in the neural networks

    # Momentum update for each parameter
    for l in range(L):


        # compute velocities
        v["dW" + str(l+1)] = beta * v['dW' + str(l+1)] + (1-beta)*(grads['dW' +
                str(l+1)])
        v["db" + str(l+1)] = beta * v['db' + str(l+1)] + (1-beta)*(grads['db' +
                str(l+1)])
        # update parameters
        parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learning_rate
               * v['dW' + str(l+1)]
        parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learning_rate
               * v['db' + str(l+1)]

    return parameters, v
```

4) Update parameters using Adam gradient descent

```
def initialize_adam(parameters) :
    """
    inputs:
    parameters -- python dictionary containing parameters.

    Returns:
```

```
    v -- python dictionary for exponentially weighted average of the gradient.
               - keys: "dW1", "db1", ..., "dWL", "dbL"
               - values: zeros with the shape as the corresponding parameters.
    s -- python dictionary for exponentially weighted average of the squared
gradient.
               - keys: "dW1", "db1", ..., "dWL", "dbL"
               - values: zeros with the shape as the corresponding parameters.

    """

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}
    s = {}

    # Initialize v, s. Input: "parameters". Outputs: "v, s".
    for l in range(L):
        v["dW" + str(l+1)] = np.zeros(parameters['W' + str(l+1)].shape)
        v["db" + str(l+1)] = np.zeros(parameters['b' + str(l+1)].shape)
        s["dW" + str(l+1)] = np.zeros(parameters['W' + str(l+1)].shape)
        s["db" + str(l+1)] = np.zeros(parameters['b' + str(l+1)].shape)

    return v, s
```

```
def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate =
0.01,beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):
    """
    Update parameters using Adam

    Arguments:
    parameters -- python dictionary containing current parameters:
    grads -- python dictionary containing gradients for each parameters:
    v -- moving average of the first gradient
    s -- moving average of the squared gradient
    t -- iteration index
    learning_rate -- the learning rate, scalar.
    beta1 -- Exponential decay hyperparameter for the first moment estimates
    beta2 -- Exponential decay hyperparameter for the second moment estimates
    epsilon -- a small positive number

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- moving average of the first gradient
    s -- moving average of the squared gradient
    """

    L = len(parameters) // 2       # number of layers in the neural networks
    v_corrected = {}   # Initializing first moment estimate, python dictionary
    s_corrected = {}   # Initializing second moment estimate, python dictionary

    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs:"v,grads,beta1".Output: "v".
        v["dW"+str(l+1)]=beta1*v['dW'+str(l+1)]+(1-beta1)*grads['dW'+str(l+1)]
        v["db"+str(l+1)]=beta1*v['db'+str(l+1)]+(1-beta1)*grads['db'+str(l+1)]

        # Compute bias-corrected first moment estimate. Inputs: "v,beta1,t".
        # Output: "v_corrected".
        v_corrected["dW"+str(l+1)]=v['dW'+str(l+1)]/(1-np.power(beta1,t))
        v_corrected["db"+str(l+1)]=v['db'+str(l+1)]/(1-np.power(beta1,t))

        # Moving average of the squared gradients.
        # Inputs: "s, grads, beta2". Output: "s".
        s["dW"+str(l+1)]=beta2*s['dW'+str(l+1)]+(1-beta2)*np.power(grads['dW' +
str(l+1)],2)
        s["db"+str(l+1)]=beta2*s['db'+str(l+1)]+(1-beta2)*np.power(grads['db' +
str(l+1)],2)

        # Compute bias-corrected second raw moment estimate.
        # Inputs: "s, beta2, t". Output: "s_corrected".
        s_corrected["dW" + str(l+1)] = s['dW' + str(l+1)] / (1 -
np.power(beta2, t))
        s_corrected["db" + str(l+1)] = s['db' + str(l+1)] / (1 -
np.power(beta2, t))

        # Update parameters.
        # Inputs: "parameters,learning_rate,v_corrected,s_corrected, epsilon".
        # Output: "parameters".
        parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learning_rate
* v_corrected['dW' + str(l+1)] / np.sqrt(s_corrected['dW' + str(l+1)] +
epsilon)
        parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learning_rate
* v_corrected['db' + str(l+1)] / np.sqrt(s_corrected['db' + str(l+1)] +
epsilon)

    return parameters, v, s
```

5) Generate batches for one epoch in a random way.

```
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples)
    Y -- true "label" vector, of shape (1, number of examples)
    mini_batch_size -- size of the mini-batches, integer

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    np.random.seed(seed)        # To make your "random" minibatches deterministic
    m = X.shape[1]              # number of training examples
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((Y.shape[0],m))

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size)
    # number of mini batches of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):

        mini_batch_X = shuffled_X[:, k*mini_batch_size : (k+1)*mini_batch_size]
        mini_batch_Y = shuffled_Y[:, k*mini_batch_size : (k+1)*mini_batch_size]

        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

        # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[:, num_complete_minibatches*mini_batch_size:]
        mini_batch_Y = shuffled_Y[:, num_complete_minibatches*mini_batch_size:]

        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

6) Parameter initialization

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

    Returns:
    parameters -- python dictionary containing parameters
                  Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                  bl -- bias vector of shape (layer_dims[l], 1)

    """

    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(2 / 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
```

7) Forward propagation

```
def forward_propagation(X, parameters):
    """
    Implements the forward propagation.

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

    Returns:
    a3 -- output of neural network
    cache -- internal signals and parameters
    """

    # 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 -> SOFTMAX
    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 = softmax(z3)

    cache = (z1, a1, W1, b1, z2, a2, W2, b2, z3, a3, W3, b3)

    return a3, cache
```

8) Compute cost

```
def compute_cost(a3, Y):

    """
    Implement the cost function

    Arguments:
    a3 -- post-activation, output of forward propagation
    Y -- "true" labels vector, same shape as a3

    Returns:
    cost - value of the cost function
    """
    m = Y.shape[1]
    k = Y.shape[0]
    if k==1: # sigmoid for 2 classes
        logprobs = np.multiply(-np.log(a3),Y) + np.multiply(-np.log(1 - a3), 1 - Y)
        cost = 1./m * np.sum(logprobs)

    else:  #softmax for multiple classes, each column in Y is one-hot-code
        logprobs = np.multiply(np.log(a3),Y)
        cost = -1/m*np.sum(logprobs)
        cost = np.squeeze(cost)
    return cost
```

9) Backward propagation

```
def backward_propagation(X, Y, cache):
    """
    Implement the backward propagation.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector
    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 = 1./m * (a3 - Y)
    dW3 = np.dot(dz3, a2.T)
    db3 = np.sum(dz3, axis=1, keepdims = True)

    da2 = np.dot(W3.T, dz3)
    dz2 = np.multiply(da2, np.int64(a2 > 0))
    dW2 = np.dot(dz2, a1.T)
    db2 = np.sum(dz2, axis=1, keepdims = True)

    da1 = np.dot(W2.T, dz2)
    dz1 = np.multiply(da1, np.int64(a1 > 0))
    dW1 = np.dot(dz1, X.T)
    db1 = 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
```

10) Put all together: build the training model

```
def model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size =
64, beta = 0.9, beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000,
print_cost = True):
    """
    3-layer neural network model which can be run in different optimizer modes.

    Arguments:
    X -- input data, of shape (2, number of examples)
    Y -- true "label" vector, of shape (1, number of examples)
    layers_dims -- python list, containing the size of each layer
    learning_rate -- the learning rate, scalar.
    mini_batch_size -- the size of a mini batch
    beta -- Momentum hyperparameter
    beta1 -- Exponential decay hyperparameter for the past gradients estimates
    beta2 -- Exponential decay hyperparameter for the past squared gradients
estimates
    epsilon -- hyperparameter preventing division by zero in Adam updates
    num_epochs -- number of epochs
    print_cost -- True to print the cost every 1000 epochs

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    L = len(layers_dims)     # number of layers in the neural networks
    costs = []               # to keep track of the cost
    t = 0                    # initializing the counter required for Adam update
    seed = 10                # For repeatability

    # Initialize parameters
    parameters = initialize_parameters(layers_dims)
     # Initialize the optimizer
    if optimizer == "gd":
        pass # no initialization required for gradient descent
    elif optimizer == "momentum":
        v = initialize_velocity(parameters)
    elif optimizer == "adam":
        v, s = initialize_adam(parameters)

    # Optimization loop
    for i in range(num_epochs):

        # Define the random minibatches.
        # We increment the seed to reshuffle differently the dataset
        # after each epoch
        seed = seed + 1
        minibatches = random_mini_batches(X, Y, mini_batch_size, seed)

        for minibatch in minibatches:

            # Select a minibatch
            (minibatch_X, minibatch_Y) = minibatch

            # Forward propagation
            a3, caches = forward_propagation(minibatch_X, parameters)

            # Compute cost
            cost = compute_cost(a3, minibatch_Y)
         # Backward propagation
            grads = backward_propagation(minibatch_X, minibatch_Y, caches)

            # Update parameters
            if optimizer == "gd":
                parameters = update_parameters_with_gd(parameters, grads, 
learning_rate)
            elif optimizer == "momentum":
                parameters, v = update_parameters_with_momentum(parameters, 
grads, v, beta, learning_rate)
            elif optimizer == "adam":
                t = t + 1 # Adam counter
                parameters, v, s = update_parameters_with_adam(parameters, grads, 
v, s, t, learning_rate, beta1, beta2,  epsilon)

        # Print the cost every 1000 epoch
        if print_cost and i % 100 == 0:
            #print(t)
            print ("Cost after epoch %i: %f" %(i, cost))
        if print_cost and i % 10 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('epochs (per 10)')
    plt.title("Learning rate = " + str(learning_rate))
    plt.show()

    return parameters, costs
```

11) Convert the class labels to one-hot codes. Note that the label to the model was defined as one-hot code. Thus, we need to convert the label (e.g. an integer) to one-hot code before calling the training model.

```
def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)].T
    return Y
```

12) Performance evaluation: predict accuracy, and plot decision boundary.

```
def predict(X, parameters):
    """
    This function is used to predict the results of an n-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model
    Returns:
    y_esti -- predictions (labels:0,1,2,3,4,,5) for the given dataset X
    """

    m = X.shape[1]
    y_esti = np.zeros((1,m), dtype = np.int)
    # Forward propagation
    a3, caches = forward_propagation(X, parameters)


    for i in range(0, a3.shape[1]):
        y_esti[0,i]=np.argmax(a3[:,i])

    return y_esti
```

```
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y.ravel(), cmap=plt.cm.Spectral)
    plt.show()
```

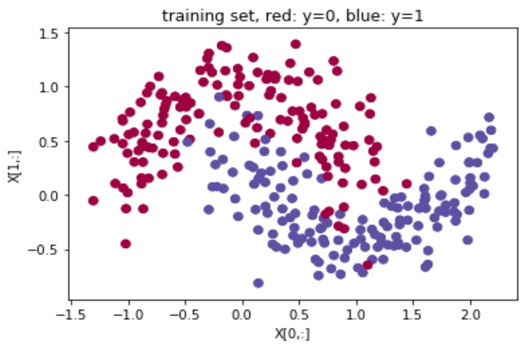
```
def predict_dec(parameters, X):
    """
    Used for plotting decision boundary.

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (m, K)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """

    # Predict using forward propagation and a classification threshold of 0.5
    a3, cache = forward_propagation(X, parameters)
    predictions = (a3 > 0.5)
    return predictions
```

13) Train the model on the *Moon* dataset for a binary classification:

Generate the training data

```
def load_dataset():
    np.random.seed(3)
    train_X, train_Y = sklearn.datasets.make_moons(n_samples=300, noise=.2) #300 #0.2
    # Visualize the data
    plt.scatter(train_X[:,0],train_X[:,1],c=train_Y.ravel(),s=40, cmap=plt.cm.Spectral);
    train_X = train_X.T
    train_Y = train_Y.reshape((1, train_Y.shape[0]))

    return train_X, train_Y
```

```
train_X, train_Y = load_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("training set, red: y=0, blue: y=1")
plt.show()
```



Train the model

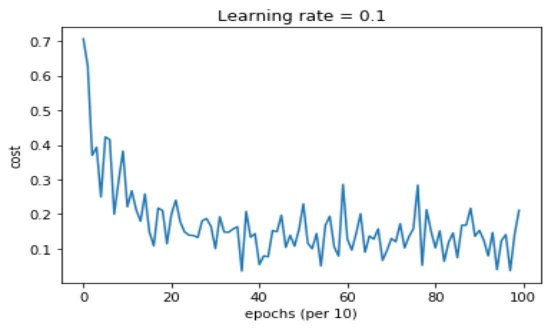
```
layers_dims = [train_X.shape[0], 5, 2, 2]
train_Y_hot = convert_to_one_hot(train_Y, 2)
#     model(X, Y, layers_dims, optimizer, learning_rate = 0.0007,
#     mini_batch_size = 64, beta = 0.9,
#     beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000,
#     print_cost = True)
#

parameters, costs=model(train_X, train_Y_hot, layers_dims,
optimizer="momentum", learning_rate = 0.1, mini_batch_size = 64, beta = 0.9,
beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 1000, print_cost =
True)

# Predict
predictions = predict(train_X, parameters)
accuracy = np.mean(predictions == train_Y)

print("accuracy of train set is "+ str(accuracy))
# Predict
#predictions = predict(train_X, train_Y, parameters)
 # Plot decision boundary
plt.title("Model with momentum optimization")
axes = plt.gca()
axes.set_xlim([-1.5,2.5])
axes.set_ylim([-1,1.5])
plot_decision_boundary(lambda x: predict(x.T, parameters), train_X, train_Y)
```

Performance evaluation

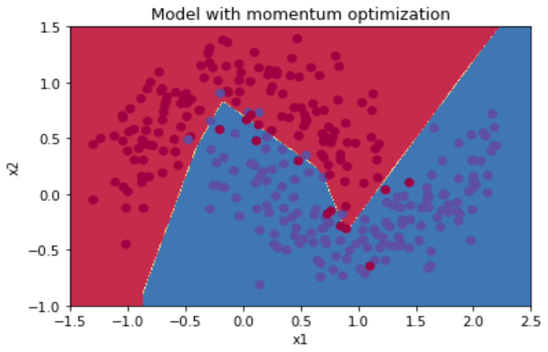
```
# Predict
```

```python
predictions = predict(train_X, parameters)
accuracy = np.mean(predictions == train_Y)

print("accuracy of train set is "+ str(accuracy))
# Predict
#predictions = predict(train_X, train_Y, parameters)
 # Plot decision boundary
plt.title("Model with momentum optimization")
axes = plt.gca()
axes.set_xlim([-1.5,2.5])
axes.set_ylim([-1,1.5])
plot_decision_boundary(lambda x: predict(x.T, parameters), train_X, train_Y)
```

```
Cost after epoch 0: 0.706019
Cost after epoch 100: 0.221365
Cost after epoch 200: 0.199276
Cost after epoch 300: 0.100450
Cost after epoch 400: 0.054298
Cost after epoch 500: 0.229595
Cost after epoch 600: 0.127700
Cost after epoch 700: 0.129801
Cost after epoch 800: 0.102676
Cost after epoch 900: 0.152990
```



```
accuracy of train set is 0.94
```



14) Train the model on MNIST dataset for a 10-class classification. The MNIST is a popular benchmark dataset and was described in Section 5.7.2. In summary, the dataset consists of 60000 examples for training and 10000 examples for testing. Each example is a 28x28 gray image for one of ten handwritten digits. Each pixel value is represented by an integer in the range of [0, 255].

As an example, we choose the architecture, shown in Fig.6.24, for the neural network. The input of neural network is a gray image of 28x28, which has 784 features since each pixel corresponds to one feature.

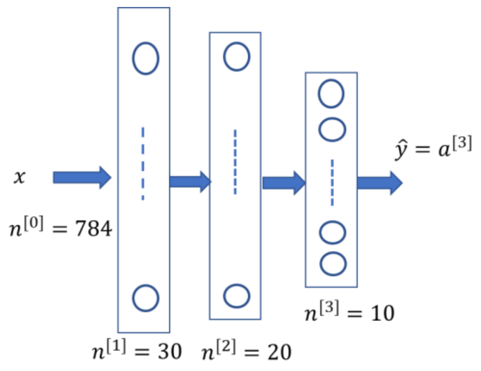


Fig.6.24 Neural network for image classification

<u>Prepare data:</u>

Load data

```
train_data = np.loadtxt("C:/machine_learning/NN_nn_overview/mnist_train.csv",
                        delimiter=",")
test_data = np.loadtxt("C:/machine_learning/NN_nn_overview/mnist_test.csv",
                       delimiter=",")

fac = 0.99/255 # convert [0,255] to [0,1]
train_imgs = np.asfarray(train_data[:, 1:])*fac+0.01
test_imgs = np.asfarray(test_data[:, 1:])*fac+0.01
# first column for labels

train_labels = np.asfarray(train_data[:,:1])
test_labels = np.asfarray(test_data[:, :1])
```

Convert data to the format for the neural network

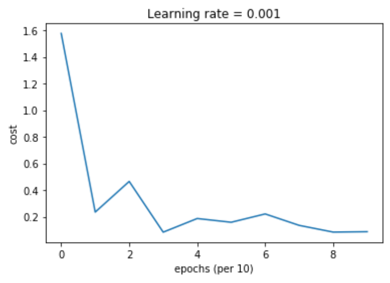
```
train_imgs_T=np.transpose(train_imgs)   # shape to (784, 60000)
test_imgs_T=np.transpose(test_imgs)     # shape to (784, 10000)
train_labels_T=np.transpose(train_labels)  # shape to (1, 60000)
test_labels_T=np.transpose(test_labels)  # shape to (1, 10000)

# convert labels to one-hot codes
train_Y_hot = convert_to_one_hot(train_labels_T.astype(int), 10)
# shape (10, 60000)
test_Y_hot = convert_to_one_hot(test_labels_T.astype(int), 10)
# shape (10, 10000)
```

<u>Train the model</u>

```
# train 3-layer model
layers_dims = [train_imgs_T.shape[0], 30, 20, 10]
parameters, costs=model(train_imgs_T, train_Y_hot, layers_dims, optimizer="gd",
learning_rate = 0.001, mini_batch_size = 32, beta = 0.9,
 beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 100, print_cost =
True)
```

Performance: accuracy

```
# Predict
predictions = predict(train_imgs_T, parameters)
accuracy = np.mean(predictions == train_labels_T)

print("accuracy of train set is "+ str(accuracy))

predictions = predict(test_imgs_T, parameters)
accuracy = np.mean(predictions == test_labels_T)
print("accuracy of test set is "+ str(accuracy))
```



```
accuracy of train set is 0.95945
accuracy of test set is 0.9558
```

Please note that one can use different values for hyperparameter when training the model. For example, you can use a different optimizer, learning rate, batch size, and so on.

Summary

In this chapter, we present some important practical considerations while developing a neural network. These considerations include overfitting, 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 the major considerations (regularization, dropout, parameter initialization, optimization, mini-batch gradient descent) in python programming.

To fully understand the principles, we implemented the examples from scratch. However, it is common practice to use python packages and design frameworks (e.g. TensorFlow and PyTorch) to develop a real project more efficiently and reliably, especially when the neural networks are becoming deeper. From the next chapter, we introduce PyTorch framework, and apply it to deep neural networks in subsequent chapters.

Files:

C:/Users/weido/ch4_practical/regularization_dropout.ipynb

C:/Users/weido/ch4_practical/data.mat

C:/Users/weido/ch4_practical/mini_batch.ipynb

C:/machine_learning/NN_nn_overview/mnist_train.csv

C:/machine_learning/NN_nn_overview/mnist_test.csv

Further reading

For dropout

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

For initialization

[2] Glorot X, Bengio Y (2010), Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics, pp 249–256.

[3] He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE international conference on computer vision, pp 1026–1034.

For batch normalization

[4] Sergey Ioffe, Christian Szegedy (2015), Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167 **[cs.LG]**

For adam optimization

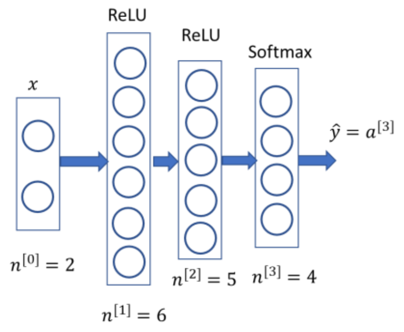
[5] Diederik P. Kingma, Jimmy Ba (2015), Adam: A Method for Stochastic Optimization. The 3rd International Conference for Learning Representations, San Diego, 2015. arXiv:1412.6980 **[cs.LG].**

Exercises

1. Show that inserting a linear layer (i.e. the activation function is a unit function) between two layers of a neural network never increases the expressive power of the neural network.

2. Suppose that our trained neural network achieves an accuracy of 0.98 on the training set and an accuracy of 0.71 on the testing set. Thus, the model is likely overfitting. Discuss the possible solutions to the overfitting.

3. Consider a neural network below and refer to the examples in Section 6.9.



1) Write the forward propagation equations and specify the shapes of all variables and parameters.

2) Write the derivative backpropagation equations and specify the shapes of all variables.

3) Write Python functions for ReLU and Softmax.

```
def softmax(x):


def relu(x):
```

4) Write a Python function to initialize all the parameters using He intitialization:

```
def initialize_parameters(layer_dims):

    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer
        layer_dims = [2, 6, 5, 4] for this neural network

    Returns:  He initialization
        parameters -- python dictionary containing parameters "W1", "b1", ...,
                      "WL", "bL":
        Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
        bl -- bias vector of shape (layer_dims[l], 1)

    """
```

5) Write a Python function to implement the forward propagation.

```
def forward_propagation(X, parameters):
    """
    Implements the forward propagation.

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

    Returns:
    a3 -- output of neural network
    cache -- internal signals and parameters
    """
```

6) Write a Python function to implement the backward propagation.

```
def backward_propagation(X, Y, cache):
    """
    Implement the backward propagation.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector
    cache -- cache output from forward_propagation()
    Returns:
    gradients -- A dictionary with the gradients with respect to each
                 parameter, activation and pre-activation variables
    """
```

7) Write a Python program that initializes the model parameters, compute one forward propagation and one backward propagation, and one parameter update with learning rate =0.1, and two examples: $x^{(1)} = \begin{pmatrix}1\\2\end{pmatrix}, y^{(1)} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, x^{(2)} = \begin{pmatrix}-1\\0\end{pmatrix}, y^{(2)} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$.

4. For the example in Section 6.9.1, let's optimize the hyperparameters jointly. Find the best set of (lambd, keep_prob, learning_rate) for the dataset, by trying the different value combinations.

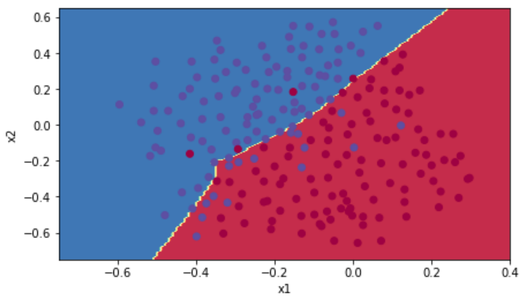
```
model(X,Y, learning_rate=0.3, num_iterations=30000, print_cost= True,
lambd=0, keep_prob=1)
```

5. In Section 6.9.1 (*regularization_dropout.ipynb*), 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 *regularization_dropout.ipynb* 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)

6. The MNIST training set has 60000 examples. If we set the mini-batch size to be 64, and the training process runs for 100 epochs, then how many updates will be applied to parameters in total?
7. The neural network in Fig.6.24 achieves an accuracy of 95.5% on the MNIST test set, with the following settings:

```
# train 3-layer model
layers_dims = [train_imgs_T.shape[0], 30, 20, 10]
parameters, costs=model(train_imgs_T, train_Y_hot, layers_dims, optimizer="gd",
learning_rate = 0.001, mini_batch_size = 32, beta = 0.9,
 beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 100, print_cost =
True)
```

Can you improve the accuracy by changing the settings including layers_dims and other hyperparameters?