This is the second course of Deep Learning Specialization provided by deeplearning.ai

Train/Dev/Test sets

	Application
Training set	train models
Cross-validation/Development set	test performance of different models
Test set	get an unbiased estimate for the best model

As the data set gets larger, the percentages of dev set and test set goes down. There is no need to increase dev/ test set proportionally to evaluate performance of models.

Bias and Variance

A high training set error indicates that there may be a bias error. Compared to training set error, a much higher dev set error may indicate high variance.

Regularization

- L-2 regularization for logistic regression:

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_2^2$$

- L-1 Regularization for logistic regression:

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_1$$

- Regularization for neural network:

$$\mathcal{J}(w^{[1]}, b^{[1]}, \ldots, w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{l}||w||_F^2$$

where $||w^{[l]}|| = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2$.

Dropout regularization

- Idea:

  > The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.

- Inverted dropout (forward propagation, Step 1 - 4; backward propagation, Step 5 - 6)
  1. Creat a variable $d^{[1]}$ with the same shape as $a^{[1]}$ using `np.random.rand()` to randomly get numbers between 0 and 1
  2. Set each entry of $D^{[1]}$ to be 0 with probability (1-keep_prob) or 1 with probability (keep_prob), by thresholding values in $D^{[1]}$ appropriately
  3. Set $A^{[1]}$ to $A^{[1]} * D^{[1]}$
  4. Divide $A^{[1]}$ by keep_prob
  5. In backward propagation, shut down the same neurons by reapplying the same mask $D^{[1]}$ to $dA^{[1]}$
  6. Divided $dA^{[1]}$ by keep_prob to scale $A^{[1]}$ and $dA^{[1]}$ in the same way.
- No dropout at test time
- Set different keep-probs for different layers (a higher keep-prob for layers being less worried about overfitting)
- Very frequently used in computer vision (as the input size is very large, and there is usually no suficient data to prevent overfitting)
- Cost function $\mathcal{J}$ is less well-defined (turn off dropout to make sure $\mathcal{J}$ is decreasing and then turn on dropout to train model)

- Data augmentation (flipping images, random crop, rotations or distortions of images)
- Early stopping

Normalizing Inputs

1. Subtract mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$x := x - \mu$$

2. Normalize variance

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)})^2$$
$$x := x/\sigma^2$$

Also use μ and σ^2 to normalize test set.

Vanishing / exploding gradients

Refer to: Why are deep neural networks hard to train

Weights Initialization for Deep Networks

- He initialization for ReLU

  randomly initialized w times: $\sqrt{\frac{2}{dimension\ of\ the\ previous\ layer}}$
- Initialization for Tangent

  randomly initialized w times: $\sqrt{\frac{1}{dimension\ of\ the\ previous\ layer}}$ or $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$

Gradient Checking

Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ (and $\mathcal{J}$).

Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$. ($\frac{\partial \mathcal{J}}{\partial W} = dW$)

For each i:

$$d\theta_{approx}^{[i]} = \frac{\mathcal{J}(\theta_1, \theta_2, \dots, \theta_i + \varepsilon, \dots) - \mathcal{J}(\theta_1, \theta_2, \dots, \theta_i - \varepsilon, \dots)}{2\varepsilon}$$

Check:

$$diff = \frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$$

If diff is very small (10^{-7}), there seems to be no error in the gradient.

If diff locates near 10^{-5}, we should check it carefully if there exits errors.

Tips:

- Don't use in training - only to debug
- If algorithm fails grad check, look at components to try to identify bug
- Remember regularization
- Don't work with dropout
- Run at random initialization; perhaps again after some training (if w, b are very close to zero, grads with bugs may pass grad check)

Mini-batch Gradient Descent

$x^{\{t\}}$ represents t-th mini batch of the training set.

Mini-batch size to be chosen:

- Mini-batch size = 1: stochastic gradient descent

Stochastic gradient descent loses speedup from vectorization.

- Set mini-batch size to be 2^n.
- Make sure mini-batch fits in CPU/GPu memory.

Exponentially Weighted Averages

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

v_t as approximately average over last $\frac{1}{1-\beta}$ days' temperature (last $\frac{1}{1-\beta}$ examples)

$$v_t = \sum_{i=1}^{t}(1-\beta)\beta^{t-i}\theta_i$$

Given $0 < \varepsilon = (1-\beta) < 1$,

$$(1-\varepsilon)^{\varepsilon^{-1}} \approx \frac{1}{e}$$

Bias correction:

$$\frac{v_t}{1-\beta^t}$$

Bias correction works for the early period of training.

Gradient Descent with Momentum

On iteration *t*:

Compute dW, db on the current mini-batch:

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

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

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

A common choice for hyperparameter β is 0.9 (ranging from 0.8 to 0.999).

> The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.

RMSprop

On iteration *t*:

Compute dW, db on the current mini-batch:

$S_{dW} = \beta S_{dW} + (1-\beta)dW^2$ (elementwise)

$S_{db} = \beta S_{db} + (1-\beta) db^2$ (elementwise)

$W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$, $b = b - \alpha \frac{db}{\sqrt{S_{db}}}$

In some ways, both momentum and RMSprop decrease learning on the directions which are not preferred and accelarates learning on the right direction.

Adam Optimization Algorithm

Adam: adaptive momentum estimation

Combine momentum and RMSprop together.

$v_{dW} = \beta_1 v_{dW} + (1-\beta_1) dW$, $v_{db} = \beta_1 v_{db} + (1-\beta_1) db$

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)(dW)^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2)(db)^2$

Implement bias correction:

$v_{dW}^{corrected} = \frac{v_{dW}}{1-\beta_1^t}$, $v_{db}^{corrected} = \frac{v_{db}}{1-\beta_1^t}$

$S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}$, $S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$

So,

$W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}+\varepsilon}}$, $b = b - \alpha \frac{v_{db}^{corrected}}{\sqrt{S_{db}^{corrected}+\varepsilon}}$

Hyperparameters choice:

- α: need to be tune
- β_1: 0.9
- β_2: 0.999
- ε: 10^{-8}

Learning Rate Decay

$$\alpha = \frac{1}{1 + decay\ rate * epoch\ num}$$

$$\alpha = 0.95^{epoch\ num} * \alpha_0$$

$$\alpha = \frac{k}{\sqrt{epoch\ num\ or\ mini\ batch\ num}} * \alpha_0$$

Manual decay (when learning takes a long time)

Tuning Process

According to Ng, the priorities of hyperparameters in neural network are as followed:

- Learning rate (α)
- Momentum term (β) & mini-batch size
- Layers & learning rate decay
- $\beta_1, \beta_2, \varepsilon$ (use the default value is ok)

Try random values in hyperparameter tuning (not to use a grid). Then zoom in to the space with better result to work on.

Appropriate scale for hyperparameters

- α: $r \in [a, b], \alpha = 10^r$, ie. $r \in [-4, -1]$
- β: $r \in [a, b], \beta = 1 - 10^r$ (when β is very close to 1, $\frac{1}{1-\beta}$) becomes very sensitive to small changes in β)

Batch Norm

Normalizing the intermediate values in neural network.

Given some intermediate values in NN, $z^{[l](i)}$.

$\mu = \frac{1}{m} \sum_i z^{(i)}$

$\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$

$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$

$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$.γ, β are learnable parameters, and make the inputs from a desirable distribution with mean or variance not restrited to 0 or 1.

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$ in NN.

In batch norm, $b^{[l]}$ in $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ is not needed because of β.

Batch norm with mini-batch:

On iteration t,

1. Forward propagation: in each hidden layer, use batch norm to replace $z[l]$ with $\tilde{z^{[l]}}$
2. Backward propagation: compute $dW^{[l]}, (db^{[l]}), d\beta^{[l]}, d\gamma^{[l]}$
3. Update parameters $W^{[l]}, \beta^{[l]}. \gamma^{[l]}$

> Batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on.

Batch norm at test time

At training time, $\mu^{\{i\}[l]}, \sigma^{2\{i\}[l]}$ represents μ, σ^2 of i-th mini-batch, l-th layer. Estimate μ, σ^2 using exponentially weighted average across mini-batch. (Theoratically, μ, σ^2 could be computed by running the whole training set through the final network.)

At test time, use the latest value of μ, σ^2 to compute $z_{norm}, \tilde{z}$ for single example at a time.

Softmax Regression

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

Activation function:

$t = e^{z^{[L]}}$

$a^{[L]} = g^{[L]}(z^{[L]}) = \frac{t_i}{\sum_i t_i}$

Loss function:

$$\mathcal{L}(\hat{y}, y) = -\sum_j y_j log\, \hat{y}_j$$

Cost function:

$$\mathcal{J}(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$