Coursera Deep Learning Specialization Notes: Improving Deep Neural Networks

Amir Masoud Sefidian

Version 1.0, November 2022

Contents

Preface

A couple of years ago I completed Deep Learning Specialization taught by AI pioneer Andrew Ng. I found this series of courses immensely helpful in my learning journey of deep learning. After years, I decided to prepare this document to share some of the notes which highlight key concepts I learned in the second course of this specialization, Improving Deep Neural Networks. This course teaches how to optimize your model's performance by applying many algorithms and techniques. For instance, how to tune the learning rate, the number of layers, and the number of neurons in each layer. Then regularization techniques like dropout and Batch Normalization are covered, to end with an optimization section that discusses stochastic gradient descent, momentum, RMS Prop, and Adam optimization algorithms. Notes are based on lecture videos and supplementary material provided and my own understanding of the topics.

The content of this document is mainly adapted from this GitHub repository. I have added some explanations, illustrations, and visualization to make some complex concepts easier to grasp for readers. This document could be a good reference for Machine Learning Engineers, Deep Learning Engineers, and Data Scientists to refresh their minds on the fundamentals of deep learning. Please don't hesitate to contact me via my website (sefidian.com) if you have any questions.

Happy Learning!
Amir Masoud Sefidian

1 Improving Deep Neural Networks

1.1 Training/Dev(Cross Validation (CV))/Test

- Training set is used to train the model's parameters.
- Dev(cross-validation) is used to train the model's hyperparameters and check the model's performance while in development.
- Test set is an unbiased set of data that was never seen by the model. Some teams may not use this and only a dev set instead.
- Traditionally with small data the split between these two sets would be either 70/30% (train/test(or dev)) or 60/20/20% (train/dev/test). With big data that can be something like 98/1/1%.
- Ensure that dev and test sets are from the same distribution.
- If the training data is from a different distribution than the test set, then it is recommended that the dev set should belong to the same distribution as the test set.

1.2 Bias and Variance

- High bias generally means underfitting - high error on the training set.
- High variance generally means overfitting - high error on the test set.
- Base error is the reference (e.g. human) error for the same task, and should be compared to the model to determine high variance or high bias.
- There used to be a tradeoff between these two but that is not so discussed in the scope of deep learning, because we can always increase the network or/and add more data.
- High bias and high variance can occur at the same time if the model underfits some parts of the model and overfits other parts.

Solutions for high bias:

- Bigger network (KEY) - does not cause high variance (with good regularization)
- Train longer
- Change NN architecture
- Hyperparameter search
- Increase the number of useful features

Solutions for high variance:

- More training data (KEY) - does not cause high bias
- Regularization
- Reduce model complexity
- Dropout
- Early Stopping
- Data Augmentation
- Batch Normalization

1.3 Regularization

- **Regularization** is also called **weight decay** because it causes weights to be smaller for higher values of lambda (the regularization parameter). It reduces **high variance**
- There is L2 and L1 regularization, L2 uses the squared of the weights, L1 uses only the norm and has the "advantage" of making the weights matrix sparse, though L2 is the most used in practice.
- There is usually no regularization of the bias term because it is just a constant.

  Regularization in Logistic Regression:

  L1 norm:

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

  L2 norm:

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

  Regularization in Neural Networks:

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

  where $\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{i=1}^{n^{[l]}}(W_{ij}^{[l]})^2$ (Frobenius norm)

  Gradient Descent update:

  $W^{[l]} \leftarrow (1 - \alpha\frac{\lambda}{m})W^{[l]} - \alpha \cdot (\text{Backprop Term})$
- **Intuitions**: Why does it help with reducing variance problems?
  - When $\lambda \to \infty$, it set the weight matrices $W^{[l]}$ to be reasonably close to zero. As a result, the neural network becomes a much smaller neural network. See Figure (1).

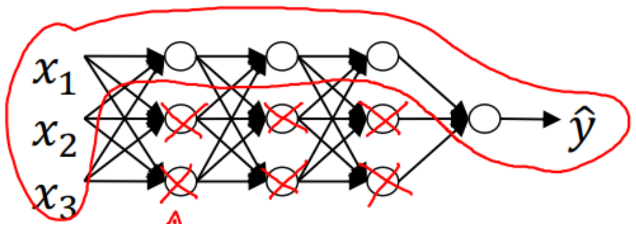


Figure 1: Regularization intuition.

  - If the regularization becomes very large, the parameters $W^{[l]} \approx 0$, so Z will be relatively small. Thus, the activation function if is *tanh*, say, will be relatively linear when $Z \to 0$. Thus, the whole neural network will be computing something not too far from a big linear function which is therefore a pretty simple function rather than a very complex highly non-linear function. See Figure (2).

1.4 Dropout

- **Dropout** regularization consists of training the NN with a number of neurons "switched off" at every training iteration (though not during testing). It has a similar effect to regularization, and it is possible to have different percentages of dropped units/neurons for each layer, making it more flexible. The same units/neurons are dropped in both forward and backward steps.

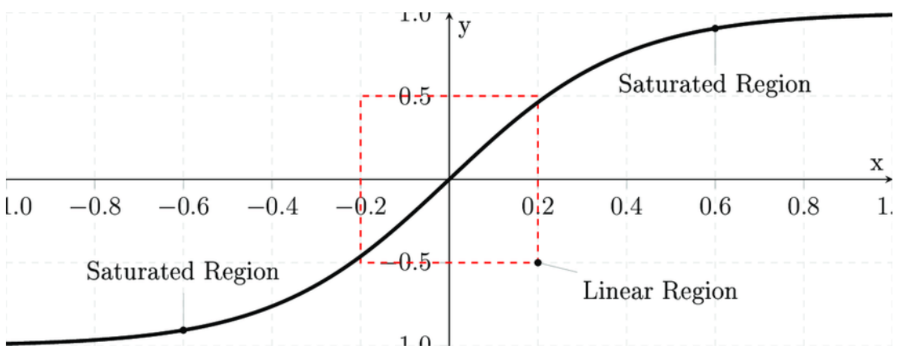


Figure 2: Regularization intuition.

The cost function with dropout does not necessarily decrease continuously as we usually see for gradient descent.

- **Inverted dropout** is the most common type of dropout, and it consists of scaling activations by dividing with the activation matrix with **keep_prob** (the probability of keeping units), for each layer.
- Steps:

Listing 1: Inverted Dropout

```
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0] a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
#ensures that the expected value of a3 remains the same
a3 /= keep_prob
```

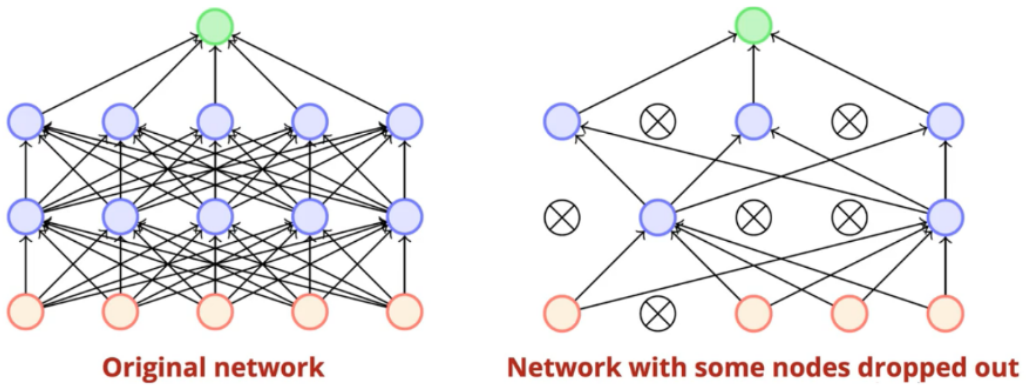


Figure 3: Dropout

- **Intuitions**:
  - Dropout randomly knocks out units in the network. Hence, it is as if on every iteration we are working with a smaller neural network, and so using a smaller neural network seems like it should have a regularizing effect.

  - Let's take a look from the perspective of a single unit. This unit takes some inputs and generates some meaningful output. Now with dropout, the inputs can get randomly eliminated. Therefore, it can't rely on any one feature because any one feature could go away at random or any one of its own inputs could go away at random. The weights, we are reluctant to put too much weight on any one input because it can go away. Thus, this unit will be more motivated to *spread out* this way and give a little bit of weight to each of inputs to this unit. And spreading all the weights will have the effect of shrinking the squared norm of the weights.

- One big downside of dropout is that the cost function J is no longer well-defined.

1.5 Other regularization methods

- **Early stopping** consists of stopping training when the error of the network is the lowest for the dev(cross-validation) dataset, even if it can still be decreased for the training set.

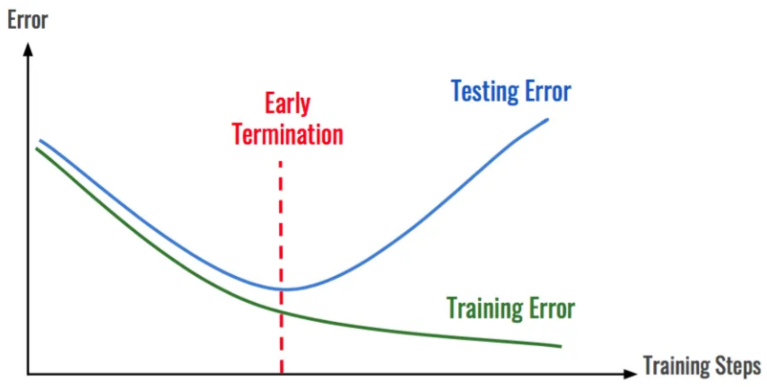


Figure 4: Early stopping

- **Data augmentation** is a technique of artificially increasing the amount of data by generating new data points from existing data. This is helpful when we are given a dataset with very few data samples. In the case of Deep Learning, this situation is bad as the model tends to overfit when we train it on a limited number of data samples. This includes adding minor alterations to data or using machine learning models to generate new data points in the latent space of original data to amplify the dataset.

- Note: **Orthogonalization** is the separation of the cost optimization step (e.g. gradient descent) from steps taken for not overfitting the model (e.g. regularization), in other words, optimizing model's parameters vs. optimizing model hyperparameters.

1.6 Normalizing training sets

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

  $x \leftarrow x - \mu$

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

  $x \leftarrow x/\sigma^2$

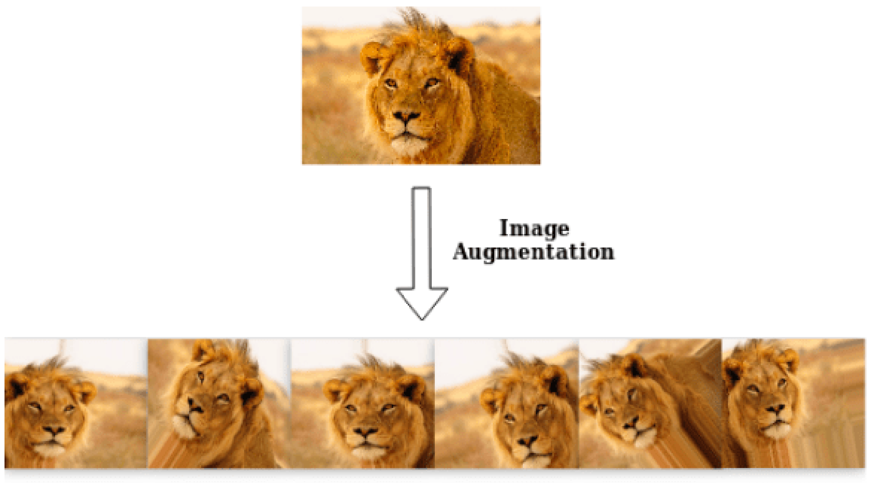


Figure 5: Data augmentation

- The mean and variance obtained in the training set should be used to scale the test set as well, (we don't want to scale the training set differently).
- Allows using higher learning rates and faster convergence for gradient descent.
- If features are on very different scales, say the feature x_1 ranges from 1 to 1000, and the feature x_2 ranges from 0 to 1, then the ratio or the range of values for the parameters w_1 and w_2 will end up taking on very different values. Then the cost function can be very elongated. When normalizing the features, the cost function will be more symmetric. When contours are spherical, we can take much larger steps with gradient descent rather than needing to oscillate. See Figure (6).

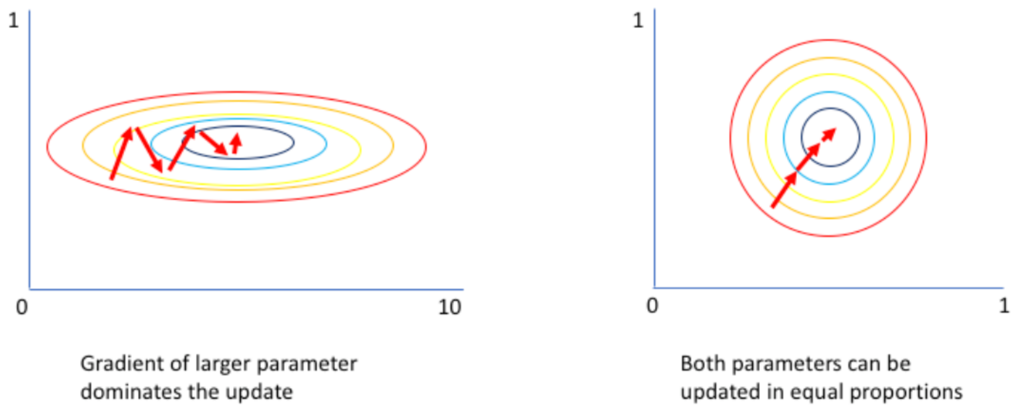

1
0
10
Gradient of larger parameter dominates the update
1
0
1
Both parameters can be updated in equal proportions

Figure 6: Normalization effect

1.7 Vanishing / Exploding gradients

- In very deep networks (depending on the activation function) weights greater than 1 can make activations exponentially larger depending on the number of layers with such weights, whereas weights smaller than 0 can make activations exponentially smaller, depending on the number of layers with such small weights (think in terms of a very deep network with linear activations as intuitive example).

- The above is also applicable for the gradients (not just the activation/output), on the opposite direction (backward propagation), thus gradients can either explode (causing numerical instability) or become very small (with the consequence of lower layers not being updated as well as numerical instability).

1.8 Weight initialization for deep networks

- Partial solution to Vanishing/Exploding gradients
- Make the randomly initialized weights to have a variance of $\frac{1}{n^{[l-1]}}$ for *tanh* and $\frac{2}{n^{[l-1]}}$ for ReLU. $n^{[l-1]}$ is the number of inputs of layer l.
- For *tanh*:
  np.random.randn() * $\sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization) or np.random.randn() * $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$
  For ReLU:
  np.random.randn() * $\sqrt{\frac{2}{n^{[l-1]}}}$

1.9 Numerical approximation of gradients

- Two-sided difference approximates the derivative of a function with $O(\epsilon^2)$ error therefore much better than the one-sided difference that is $O(\epsilon)$ - and for ϵ smaller than 0 that that means that the two-sided difference has a much smaller error.

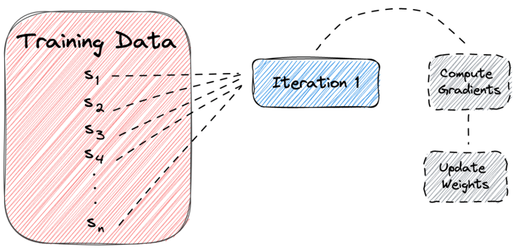
1.10 Gradient checking (Grad check)

- Used to check the correctness of the implementation (bugs). Only to be used during debugging, not during training (it's slow).
- Take $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ and concatenate and reshape them into a vector θ.
- Take $dW^{[1]}, db^{[1]}, \cdots, dW^{[L]}, db^{[L]}$ and concatenate and reshape them into a vector $d\theta$.
- With Θ being a vector of parameters θ_i, and $d\theta[i] = \frac{\partial J}{\partial \theta_i}$, compare the "approx" derivative with the real $d\theta$ with the check:
  $$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$$
  , where
  $d\theta_{approx}[i] = \frac{J(\theta_1,\theta_2,\cdots,\theta_i+\epsilon,\cdots)-J(\theta_1,\theta_2,\cdots,\theta_i-\epsilon,\cdots)}{2\epsilon}$
- Note that $||\cdot||_2$ denotes the squared root of the sum of the squared differences (that is, the norm of the vector).
- If the result is near 10^{-7} it is great, if it is 10^{-5}, then suspect something in the formula, if 10^{-3}, something is really wrong.
- Look for what $d\theta[i]$ (what component) has the highest difference, to pinpoint the cause of the bug.
- Include the regularization term in the cost function when performing **grad check**.

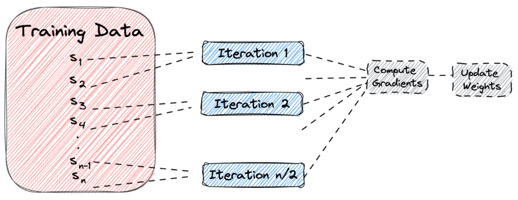
- Doesn't work with dropout (turn it off during grad check).
- Run at initialization and then again after some training.

1.11 Mini-batch gradient descent

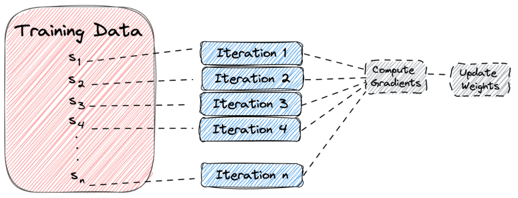
- Applicable for large datasets (single batch).
- Running each iteration of gradient descent on smaller batches of the full dataset. May take too long per iteration.
- The cost function trends downward but not monolithically in this case.
- If batch size = m then it is just batch gradient descent (run for all the examples at once). (use this for m <= 2000).
- If batch size = 1 then it is **Stochastic Gradient Descent** (every example is a mini-batch). Gradient descent doesn't completely converge. Loses speedup from vectorization.
- Ideal scenario is in between the two above (may not exactly converge, but we can reduce the learning rate).
- Typical minibatch sizes are powers of two (64, 128, 256, 512) - to ensure they fit in cpu/gpu memory.



(a) Batch GD



(b) Mini Batch GD



(c) Stochastic GD

Figure 7: Batch Gradient Descent vs. Mini Batch Gradient Descent vs. Stochastic Gradient Descent

1.12 Optimization algorithms - exponentially weighted moving averages

- $V_t = \beta V_{t-1} + (1-\beta)\theta_t$. V_t is average over past $1/(1-\beta)$ data points
- All the coefficients add up to 1.

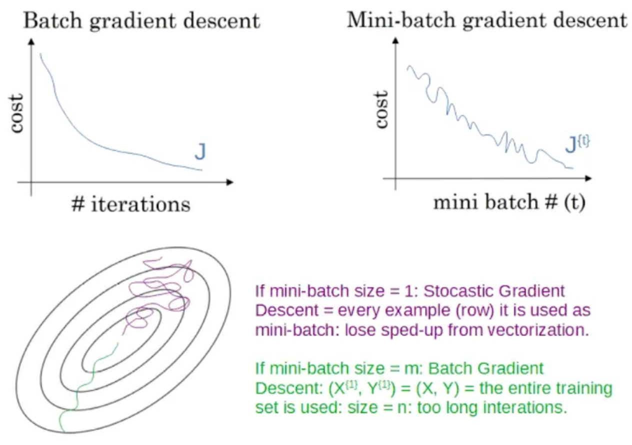


Figure 8: Batch GD cost curve

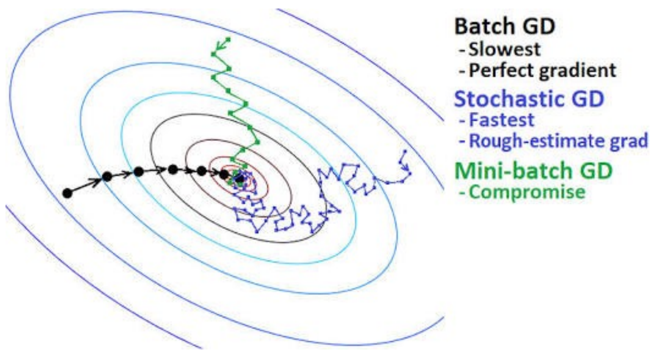


Figure 9: Convergence of different GD methods

- When $\beta = 0.9$ it takes a delay of approx 10 (think 10 days for a daily time-series data) for the contribution of a point to reduce to 1/3. The general rule (in which ϵ is 0.1 and $\beta = 1 - \epsilon$ to meet this example) is:

$$(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$$

- To correct the bias of the first few terms (compared to 0 initialization), the following formula can be used:

$$V_t^{corrected} = \frac{V_t}{1-\beta^t}$$

1.13 Gradient descent with momentum

- Converges faster than the standard gradient descent algorithm.
- The basic idea is to compute an exponentially weighted average of gradients, and then use that gradient to update the weights.
- Uses exponential weighted moving averages to smooth out the derivatives dW and db, when updating W and b in each iteration. For example for dW (and similarly for db):

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

$$W = W - \alpha V_{dW}$$

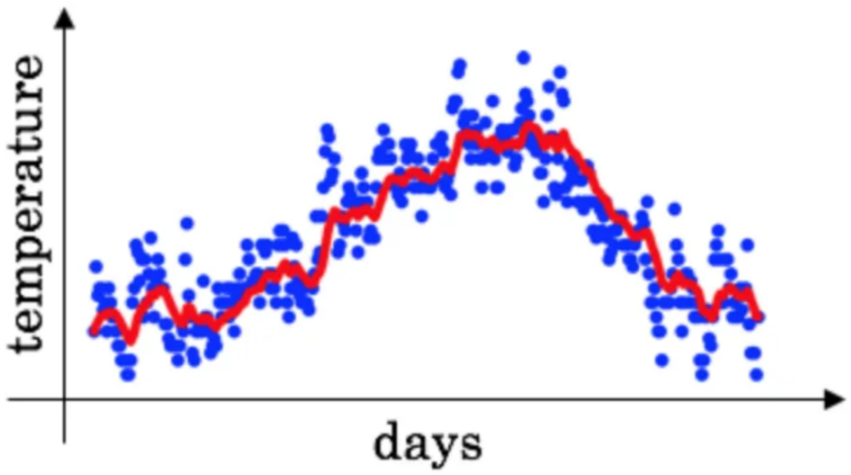


Figure 10: Exponentially Weighted Moving Average

- Sometimes a simplified version is used that factors the $(1-\beta)$ term into the learning rate (that must be adjusted) instead of it being explicit, therefore:

$$V_{dW} = \beta V_{dW} + dW$$

$$\alpha_{adjusted} = \alpha(1-\beta)$$

- β is most commonly 0.9 (pretty robust value)
- Bias correction is not usually used for gradient descent.

1.14 RMSprop (Root Mean Square prop)

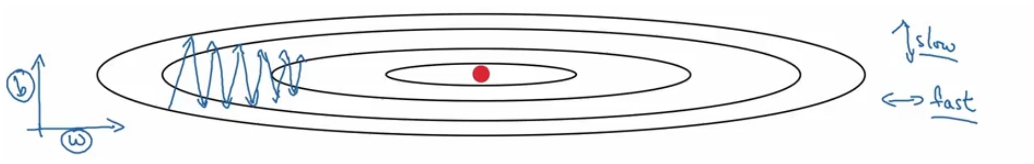


Figure 11: RMSprop.

- Update W and b on each iteration with dW or db divided by the root mean square of the exponential moving average of dW or db (the square is element-wise):

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

$$W = W - \alpha \frac{dW}{\sqrt{S_{dW} + \epsilon}}$$

- Implementations add a small ϵ to the denominator to avoid divisions by 0.
- The intuition is to have smaller/slower updates of db (derivative of the bias term or vertical direction) and higher/faster updates of dW (derivative of the weights or horizontal direction), to improve convergence speed. dW is a large matrix, therefore RMS of dW is much larger than RMS of db.
- Allows using a higher learning rate and faster convergence.

1.15 Adam (adaptive moment estimation) optimization

- Combine intuitions of Momentum + RMSprop together, both with bias correction!

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

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

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

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

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected} + \epsilon}}$$

$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$

- There are two β parameters:
  - β_1 is the momentum parameter and is usually 0.9
  - β_2 is the RMSprop parameter and is usually 0.999
  - ϵ is usually 10^{-8}

1.16 Learning rate decay (lower on the list of hyper-parameters to try)

- Have a slower learning rate as gradient descent approaches convergence.

$$\alpha = \frac{1}{1 + \text{decay\_rate} \times \text{epoch\_num}} \times \alpha_0$$

- Alternatives:
  - Exponential decay: $\alpha = 0.95^{\text{epoch\_num}} \times \alpha_0$
  - Or: $\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \times \alpha_0$
  - Discrete staircase, manual decay, etc.

1.17 Local optima and saddle points

- Most points with zero gradients are saddle points, not local optima!
  - Plateau's in saddle points slow down learning.
  - Local optima are pretty rare in comparison/unlikely to get stuck in them.

1.18 Hyperparameter tuning process

- Order of importance of hyperparameters for tuning:
    1. learning rate (α)
    2. momentum term ($\beta : 0.9$)
    3. mini-batch size
    4. number of hidden units
    5. number of layers
    6. learning rate decay
    7. $\beta_1, \beta_2, \epsilon$
- Choose the hyperparameter value combinations at random, (don't use a grid) because of the high number of hyperparameters nowadays (cube/hyper dimensional space), doesn't compensate test all values/combinations.

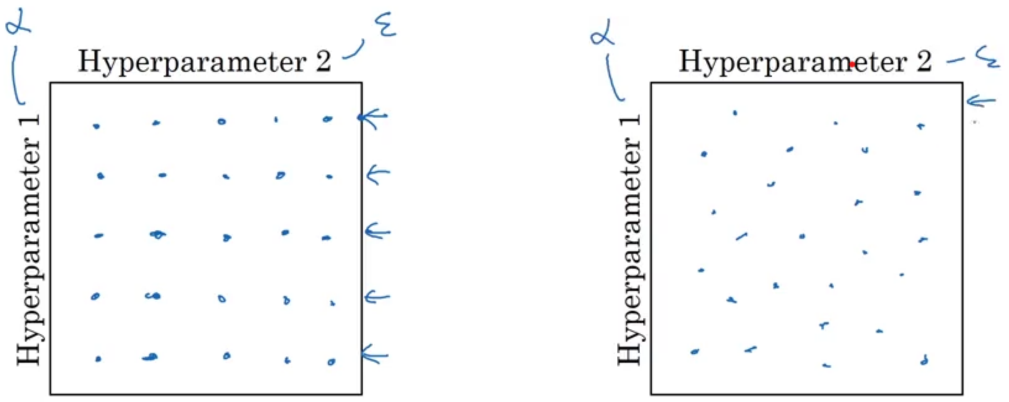


Figure 12: Grid vs. Random search.

- Coarse to fine-tuning - first coarse changes of the hyperparameters, then fine tune them.
- Using an appropriate scale for the hyperparameters.
    - One possibility is to sample values at random within an intended range.
    - Using log scales to sample parameter values to try (for example, applicable for the learning rate).
      For α (sample between $10^a \cdots 10^b$), uniformly sample from $r \in [a, b]$ ([-4, 0] for example) and set $\alpha = 10^r$.
      For exponentially weighted average hyperparameters (β, β_1, β_2), uniformly sample from $r \in [a, b]$ ([-3, -1] for example) and set $\beta = 1 - 10^r$.
- Two possible approaches for hyperparameter search:
    - Panda approach: Watch only one model and change the parameters gradually and check improvements. Requires less hardware which might not be the most efficient method.
    - Caviar approach: Run multiple models with different parameters in parallel, if you have the computing power for it.

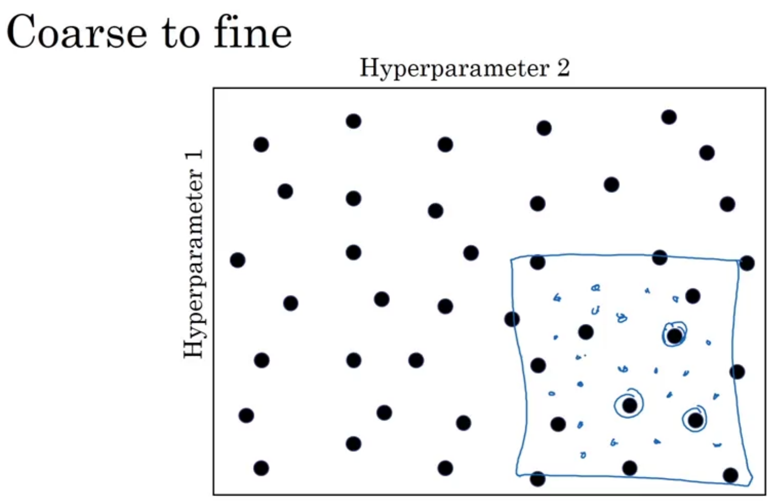


Figure 13: Coarse to fine search.

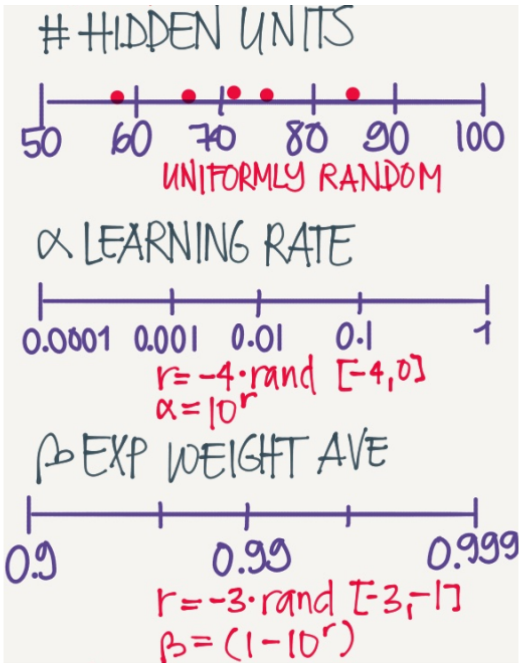


Figure 14: Hyperparameter Search

1.19 Batch normalization

- Normalize not just the inputs but the activation inputs to the next layer, subtracting the mean and dividing by the variance (mean 0, variance 1).
- Normally Z is normalized, before the activation function, though some literature suggests normalizing after the activation function.
- New parameters γ (multiplied by Z) and β (added to Z after the multiplication) are introduced and learned in the forward backward propagation. This is to prevent all neurons from having activations with mean 0 and variance 1 which is not desirable. Given some intermediate values

$z^{(1)}, z^{(2)}, \ldots, z^{(m)}$ for a layer in the network:

$$Z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \text{ where } \mu = \frac{1}{m}\sum_i z^{(i)}, \text{ and } \sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$\tilde{Z}^{(i)} = \gamma Z^{(i)}_{norm} + \beta$$

- The bias parameter "b" in the calculation of Z is no longer needed because the mean of Z is being subtracted, canceling out any effect from adding "b". The new parameter β effectively becomes the new bias term.
- At test time there is no μ and σ^2, so these are computed based on an exponentially weighted average of these two parameters obtained on different batches during training. **Why does**

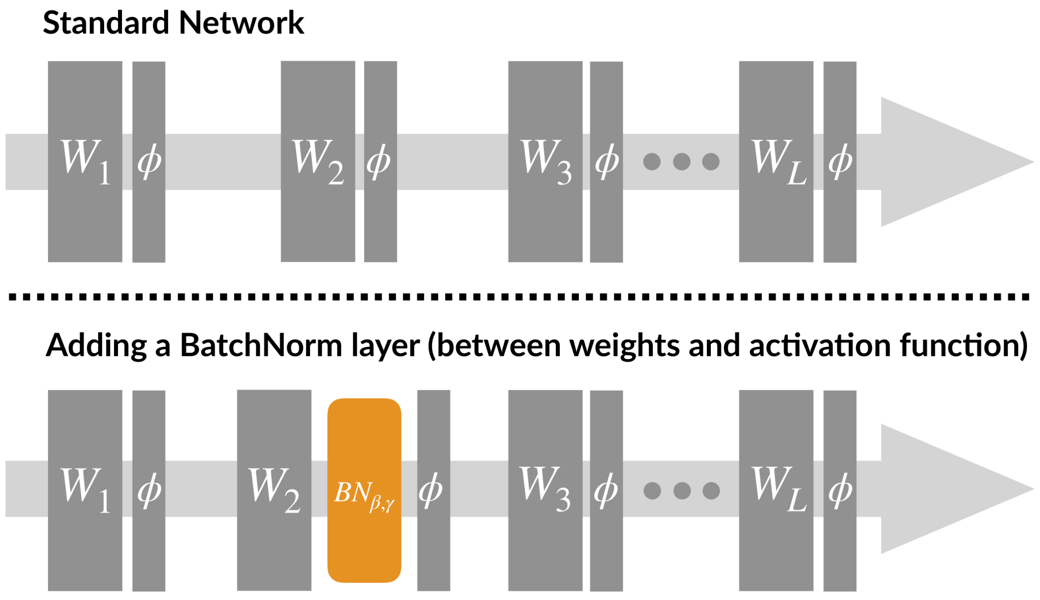

Standard Network
W_1 ϕ W_2 ϕ W_3 ϕ ● ● ● W_L ϕ
Adding a BatchNorm layer (between weights and activation function)
W_1 ϕ W_2 $BN_{\beta,\gamma}$ ϕ W_3 ϕ ● ● ● W_L ϕ

Figure 15: Batch Normalization.

Batch Normalization work?

  - It makes weights of deeper layers more robust to changes in weights in earlier layers of network.
  - **Covariate shift**: Data distribution changes with inputs (e.g. over time, with new batches, etc), you need to retrain your network normally.
  - Batch normalization makes the process of learning easier by reducing the variability of the inputs presented to each layer (which now have similar variance and mean), therefore reducing the **covariate shift**, and that is especially important for deeper layers, where inputs could change significantly as a net effect of all the other changes in the network.
  - It reduces the amount that the distribution of hidden unit values shifts around. And if it were to plot the distribution of hidden unit values, maybe this is technically we normalize Z, Batch norm ensures that no matter how the output of the previous layer changes, the mean and variance of a layer will remain the same. Therefore, the 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 have more firm ground to stand on.

- It also has a slight regularization effect with mini-batch, due to the “noise” introduced by the calculations of mean and variance only for that mini-batch only rather than the entire dataset, which has an effect similar to that of dropout.

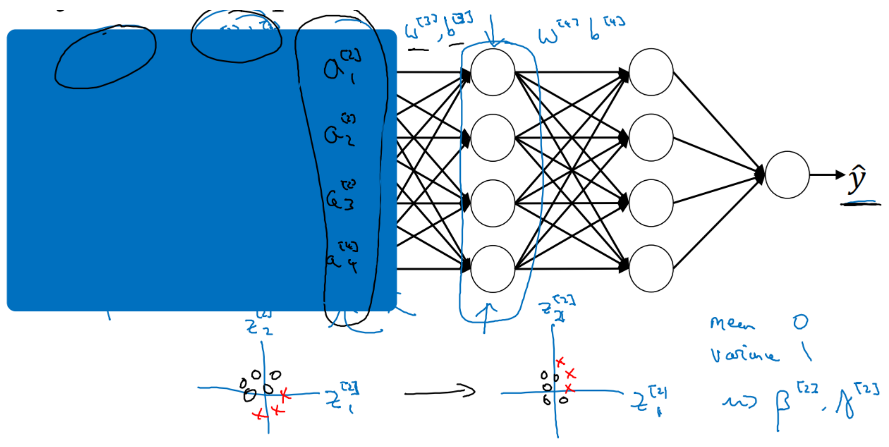


Figure 16: Covariate Shift

1.20 Multi-class classification

- # of neurons in the output layer = # of classes. The sum of all outputs must be 1, since these are probabilities of X being classified in any of these classes(likelihood).
- **Softmax** activation function is used as the output activation function:

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

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{K} t_j}, \text{ where } K \text{ is the number of classes and output units}$$

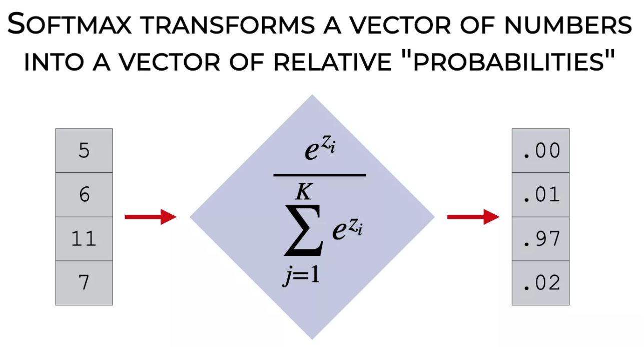

SOFTMAX TRANSFORMS A VECTOR OF NUMBERS INTO A VECTOR OF RELATIVE "PROBABILITIES"
5
6
11
7
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}$
.00
.01
.97
.02

Figure 17: Softmax

- **Softmax** is in contrast with **Hardmax**, where the network’s output will be a binary vector with all 0s except for the position corresponding to the max value of $Z^{[L]}$.

- **Softmax** is the generalization of logistic regression to more than two classes. For two classes, it can be simplified/reduced to logistic regression.

  Softmax Loss function:

  $$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j log(\hat{y}_j), \text{ assuming a binary vector } y$$

  Softmax Cost function:

  $$J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}), \text{ backprop: } \frac{\partial J}{\partial z} = \hat{y} - y$$
