Week 1: Practical Aspects of Deep Learning

Learning Objectives

- Give examples of how different types of initializations can lead to different results
- Examine the importance of initialization in complex neural networks
- Explain the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Assess the right time and place for using regularization methods such as dropout or L2 regularization
- Explain Vanishing and Exploding gradients and how to deal with them
- Use gradient checking to verify the accuracy of your back propagation implementation

Setting up your Machine Learning Application

Train/ Dev/ Test sets

Setting up the training, development (dev, also called validate set) and test sets has a huge impact on productivity. It is **important** to **choose the dev and test sets** from the **same distribution** and it **must be taken randomly** from all the data.

Guideline:

- Choose a dev set and test set to reflect data you expect to get in the future.
- The dev and test sets should be just big enough to represent accurately the performance of the model.

Bias / Variance

Error type	High Variance	High Bias	High Bias, High Variance	Low Bias, Low Variance
Train set error	1%	15%	15%	0.5%
Dev / Test set error	11%	16%	30%	1%

> *When we discuss prediction models, prediction errors can be decomposed into two main subcomponents we care about: error due to "bias" and error due to "variance". There is a tradeoff between a model's ability to minimize*

> *bias and variance. Understanding these two types of error can help use diagnose model results and avoid the mistake of over- or under-fitting.*

Bias and Variance are two sources of error in machine learning models, and they are closely related to the concepts of underfitting and overfitting.

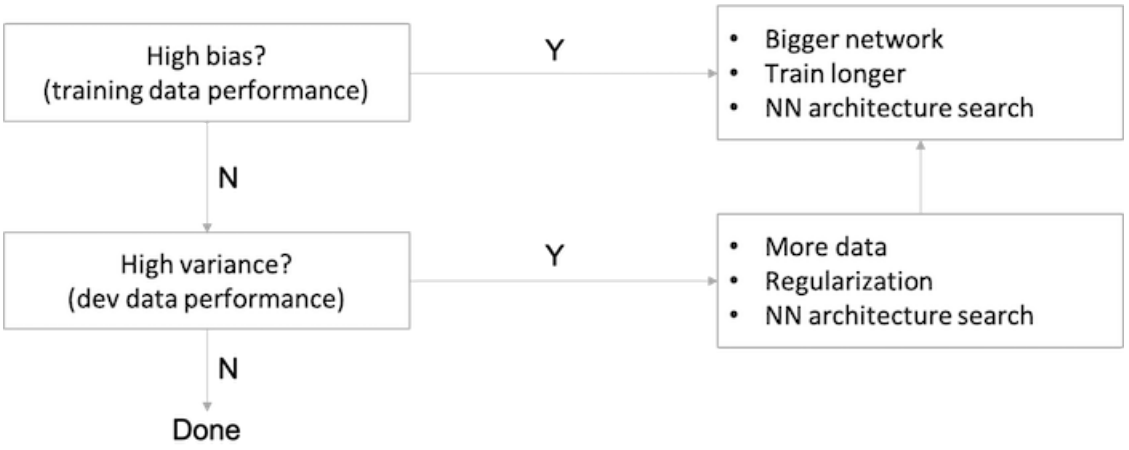
1. **Bias**:
    - **High Bias (Underfitting)**: A model with high bias pays litter attention to the training data and oversimplifies the underlying patterns. It fails to capture the complexity of the data.
    - **Low Bias**: A model with low bias fits the training data well and captures the underlying patterns effectively.
2. **Variance**:
    - **High Variance (Overfitting)**: A model with high variance fits the training data very closely but may fail to generalize well to new, unseen data. It captures noise in the training data and doesn't generalize well to variations.
    - **Low Variance**: A model with low variance generalizes well to new data and doesn't overfit the training set.

The relationship between bias/variance and underfitting/overfitting can be summarized as follows:

- **High Bias and High Variance (Underfitting and Overfitting)**: A model can suffer from both high bias and high variance simultaneously. This situation occurs when the model is too simple to capture the underlying patterns (high bias) and, at the same time, it sensitive to fluctuations in the training data (high variance).
- **Low Bias and Low Variance (Balance Model)**: An ideal model has low bias and low variance. It fits the training data well while generalizing to new, unseen data.
- **Trade-off between Bias and Variance**: There is often a trade-off between bias and variance. Increasing the complexity of a model tends to reduce bias but may increase variance, and vice versa. The goal is to find the right level of complexity that minimizes both bias and variance, leading to good generalization.
- **Bias-Variance Decomposition**: The expected prediction error of a model can be decomposed into three components: bias, variance, and irreducible error. The irreducible error is inherent noise in the data that cannot be reduced. The bias-variance trade-off involves finding a balance between bias and variance to minimize the overall expected error.

Optional Reading: Understanding the Bias-Variance Tradeoff.

Basic Recipe for Machine Learning



- For a high bias problem, getting more training data is actually not going to help.
- Back in the pre-deep learning era, we didn't have as many tools that just reduce bias or that just reduce variance without hurting the other one.
- In the modern deep learning era, getting a bigger network and more data almost always just reduces bias without necessarily hurting your variance, so long as you regularize appropriately.
- This has been one of the big reasons that deep learning has been so useful for supervised learning.
- The main cost of training a big neural network is just computational time, so long as you're regularizing.

Regularizing your Neural Network

Regularization:

Regularization for Logistic Regression:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \parallel w \parallel_2^2$$

or:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \parallel w \parallel_2^2 + \frac{\lambda}{2m} \parallel b \parallel_2^2$$

b is just one parameter over a very large number of parameters, so no need to include it in the regularization. (You can include it if you want)

Regularization	Formula	Description
L2 Regularization	$\parallel w \parallel_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$	Most common type of regularization
L1 Regularization	$\parallel w \parallel_1 = \sum_{j=1}^{n_x} \lvert w_j \rvert$	w vector will have a lot of zeros, so L1 regularization

		makes your model sparse

> **L1 and L2 regularization** are techniques used to **prevent overfitting** in machine learning models **by adding a penalty term to the cost function** associated with the model's parameters. These regularization techniques are commonly employed in linear regression, logistic regression, and neural networks, among other models.

L1 Regularization (Lasso Regularization):

L1 Regularization adds a penalty term to the cost function proportional to the absolute values of the model parameters.

Cost function with L1 Regularization:

$$J(\theta) = Original\_Cost\_Function + \lambda \sum_{j=1}^{n} |\theta_j|$$

- Original Cost Function: This is the standard cost function without regularization.
- λ: Regularization parameter. It controls the strength of the regularization. Higher values of λ lead to more regularization
- θ_j: Model parameters.
- $|\theta_j|$: Absolute value of the model parameter.

Key points:

1. **Sparse Solution**: L1 regularization tends to produce sparse solutions, meaning that it encourage some of the feature weights to be exactly zero. This can be useful for feature selection.
2. **Feature Selection**: In the context of linear regression or linear models, L1 regularization can be used for feature selection by driving some coefficient to zero.
3. **Not Robust to Outliers:** L1 regularization is not very robust to outliers because the penalty term is based on the absolute values.

L2 Regularization (Ridge Regularization)

L2 Regularization adds a penalty term to the cost function proportional to the square of the model parameters. The regularized cost function is given by:

$$J(\theta) = Original\_Cost\_Function + \lambda \sum_{j=1}^{n} \theta_j^2$$

- θ_j represents the model parameters.
- n is the number of features.

- λ is the regularization parameter, controlling the strength of regularization.

Key points:

1. **Non-Sparse Solutions:** Unlike L1 regularization, L2 regularization tends to produce non-sparse solutions. It encourages all the feature weights to be small but doesn't drive them to exactly zero.
2. **Robust to Outliers**: L2 regularization is more robust to outliers because it relies on squared values, which are less sensitive to extreme values.
3. **Shrinkage**: L2 regularization is often referred to as "weight decay" because it penalizes large weights, effectively leading to "shrinkage" of the weights.

Elastic Net Regularization:

Elastic Net combines both L1 and L2 Regularizations in the cost function:

$$J(\theta) = Original\_Cost\_Function + \lambda_1 \sum_{j=1}^{n} |\theta_j| + \lambda_2 \sum_{j=1}^{n} \theta_j^2$$

- λ_1 and λ_2 are the regularization parameters for L1 and L2 regularization, respectively.
- Elastic Net aims to provide the benefits of both L1 and L2 regularization.

Regularization formula for a Neural Network:

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

For the matrix `W`, this norm is called the Frobenius norm. Its definition looks like `L2` norm but is not called the `L2` norm:

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

Regularization of gradient:

$$dw^{[l]} = (backprop) + \frac{\lambda}{m} w^{[l]}$$

With regularization, the coefficient of `W` is slightly less than `1`, in which case it is called **weight decay**.

$$\begin{aligned} w^{[l]} &= w^{[l]} - \alpha dw^{[l]} \\ &= w^{[l]} - \alpha [(backprop) + \frac{\lambda}{m} w^{[l]}] \\ &= (1 - \frac{\alpha\lambda}{m}) w^{[l]} - \alpha (backprop) \end{aligned}$$

Why regularization reduces overfitting

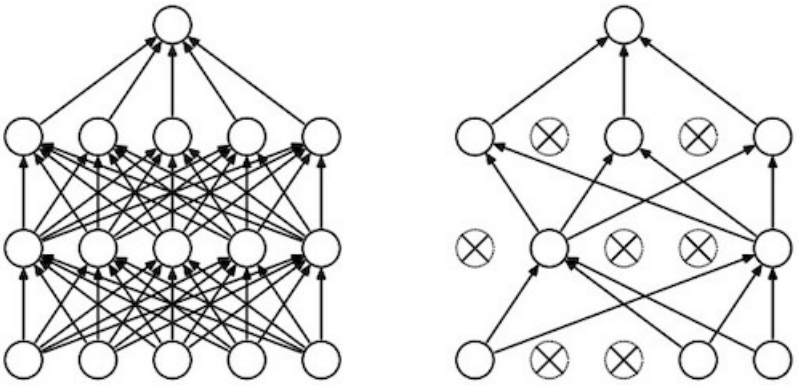
- If we make regularization lambda to be very big, then weight matrices will be set to be reasonably close to 0, effectively zeroing out a lot of hidden units. Then the simplified neural network becomes a much smaller neural network, eventually almost like a logistic regression. We'll end up with a much smaller network that is therefore less prone to overfitting.
- Taking activation function `g(z) = tanh(Z)` as example, if lambda is large, then weights `W` are small and subsequently `Z` ends up taking relatively small values, where `g` and `Z` will be roughly linear, which is not able to fit those very complicated decision boundary, i.e., less able to overfit.

Implementation tips:

Without regularization term, we should see the cost function decreases monotonically in the plot. Whereas in the case of regularization, to debug gradient descent, make sure that we plot `J` with a regularization term; otherwise, if we plot only the first term (the old `J`), we might not see a decrease monotonically.

Dropout Regularization

- Dropout is another powerful regularization technique.
- With dropout, what we're going to do is go through each of the layers of the network and set some probability of eliminating a node in neural network. It's as if on every iteration, you're working with a smaller neural network, which has a regularizing effect.
- Inverted dropout technique, `a3 = a3 / keep_prob`, ensures that the expected value of `a3` remains the same, which makes test time easier because you have less of a scaling problem.



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

Understanding Dropout

- Can't rely on any one feature, so have to spread out weights, which has an effect of shrinking the squared norm of the weights, similar to what we saw with L2 regularization, helping prevent overfitting.
- For layers where you're more worried about over-fitting, really the layers with a lot of parameters, you can set the key prop to be smaller to apply a more powerful form of drop out.
- Downside: with `keep prop` for some layers, more hyperparameters to search for using cross-validation.
- Frequently used in computer vision, as the input size is so big, inputting all these pixels that you almost never have enough data, prone to overfitting.
- Cost function `J` is no longer well-defined and harder to debug or double check that `J` is going downhill on every iteration. So first run code and make sure old `J` is monotonically decreasing, and then turn on drop out in order to make sure that no bug in drop out.

Note:

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

Other Regularization Methods

- **Data augmentation**: getting more training data can be expensive and sometimes can't get more data, so flipping horizontally, random cropping, random distortion and translation of image can make addition fake training examples.
- **Early stopping**: stopping halfway to get a mid-size `w`.
  - Disadvantage: early stopping couples two tasks of machine learning - optimizing the cost function `J` and not overfitting, which are supposed to be completely separate tasks, to make things more complicated.
  - Advantage: running the gradient descent process just once, you get to try out values of small `w`, mid-size `w`, and large `w`, without needing to try a lot of values of the L2 regularization hyperparameter lambda.

Setting up your optimization problem

Normalizing Inputs

With normalization, cost function will be more round and easier to optimize when features are all on similar scales.

- If you normalize your inputs, this will speed up the training process a lot.
- Normalization are going on these steps:
  - Get the mean of the training set: `mean = (1/m) * sum(x(i))`
  - Subtract the mean from each input: `X = X - mean`
    - This makes your inputs centered around 0.

    - Get the variance of the training set: `variance = (1/m) * sum(x(i) ^ 2)`
    - Normalize the variance: `X /= variance`
- These steps should be applied to training, dev, and testing sets (but using mean and variance of the training set).
- Why normalize?
    - If we `don't normalize` the inputs, our `cost function will be deep` and `its shape` will be `inconsistent` (elongated) then `optimizing it` will `take a long time`.
    - If we normalize it, the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Vanishing / Exploding Gradients

- The vanishing/ exploding gradients occurs when your derivatives become very small or very big.
- To understand the problem, suppose that we have a deep neural network with number of layers `L`, and **all the activation functions are linear** and each `b = 0`.
    - Then:

```
Y' = W[L]W[L-1].....W[2]W[1]X
```

    - Then, if we have 2 hidden units per layer and x1 = x2 = 1, we result in:

```
if W[l] = [1.5   0]
          [0   1.5]       (l != L because of different dimensions in the output layer)

Y' = [1.5  0]^(L-1) X  =  1.5^L   # which will be very large
     [0  1.5]
```

```
if W[l] = [0.5  0]
          [0  0.5]

Y' = [0.5  0]^(L-1) X  =  0.5^L   # which will be very small
     [0  0.5]
```

- The last example explains that the activations (and similarly derivatives) will be decreased/ increased exponentially as a function of number of layers.
- So if W > I (Identity matrix), the activation and gradients will explode.
- And if W < I (Identity matrix), the activation and gradients will vanish.
- Recently, Microsoft trained 152 layers of ResNet, which is a really big number. With such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.

- There is a partial solution that doesn't completely solve this problem but it helps a lot - careful choice of how you initialize the weights.

Weight Initialization for Deep Networks

- A partial solution to the Vanishing/ Exploding gradients in NN is better or more careful choice of the random initialization of weights.
- In a single neuron (Perceptron model): $Z = w^1x^1 + w^2x^2 + ... + w^nx^n$
  - So if `n_x` is large then we want `W`s to be smaller to not explode the cost.
- So turn out the we need the variance which equals `1/n_x` to be the range of `W`s.
- So lets say when we initialize `W`s like this (better to use with `tanh` activation):

```
np.random.rand(shape) * np.sqrt(2 / ( n[l - 1] + n[l]))
```

  or variation of this (Bengio et al.):

```
np.random.rand(shape) * np.sqrt(2 / (n[l - 1] + n[l]))
```

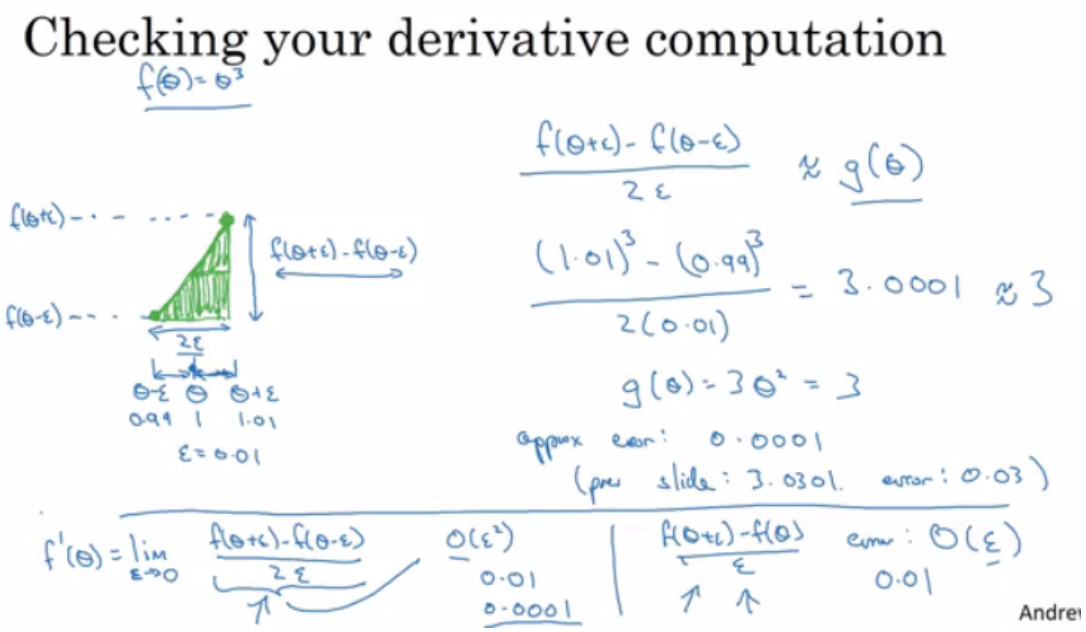
- Setting initialization part inside sqrt to `2 / n[l - 1]` for `ReLU` is better:

```
np.random.rand(shape) * np.sqrt(2 / n[l - 1])
```

- Number 1 or 2 in the numerator can also be a hyperparameter to be tuned (but not the first to start with).
- This is one of the best way of partially solution to Vanishing/ Exploding gradients (ReLU + Weight initialization with variance) which will help gradients not to vanish/explode too quickly.
- The initialization in this video is called "Xavier Initialization" and has been published in 2015 paper.

Numerical Approximation of Gradients

- There is a technique called gradient checking which tells you if your implementation of backpropagation is correct.
- There's a numerical way to calculate the derivative:



Gradient checking

- Gradient checking approximates the gradients and is very helpful for finding the errors in your backprop implementation but it's slower than gradient descent (so use only for debugging).
- **Technique**:
  - **Gradient Computation:**
    - During the training of a machine learning model, gradients of the cost function with respect to the model parameters (weights and biases) are computed. These gradients indicate how the cost function changes as you adjust the parameters, and they guide the optimization process to find the best parameter values.
  - **Numerical Gradient Approximation:**
    - Gradient checking involves approximating the gradients numerically by perturbing the parameters slightly and observing the change in the cost function. Specifically, for each parameter θ, you compute an approximate gradient $\frac{\partial J}{\partial \theta}$ by taking the difference in the cost function's values at two points: $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$
    - Hence, ϵ is a small value (often called the "epsilon" or "epsilon value") that determines how much you perturb the parameter.
  - **Comparing Gradients:**
    - You compare the gradients computed analytically (using backpropagation or other gradient calculation methods) with the gradients estimated numerically. If the analytical and numerical gradients are very close, it suggests that your gradient computation is likely correct.
  - **Threshold for Comparison:**
    - To determine if the gradients are close enough, you typically use a threshold value, often denoted as ϵ_{grad}. If the absolute difference between the analytical and

numerical gradients for each parameter is smaller than ϵ_{grad}, you consider the gradients to be consistent.

- **Checking for all the parameters:**
  - Gradient checking is performed for all parameters in your model, one parameter at a time. You iterate through all the parameters and compute the numerical gradients for each of them, comparing them to the corresponding analytical gradients.
- **Debugging and Adjustment:**
  - If the gradients don't match within the specified threshold, it indicates a potential issue with your gradient computation or backpropagation implementation. You would then need to debug and fix the problem, which could involve examining your code, checking for errors, and revisiting your model architecture.

- Gradient checking:
  - First take `W[1], b[1], ..., W[L], b[L]` and reshape it into one big vector (`theta`)
  - The cost function will be `J(theta)`
  - Then take `dW[1], db[1], ..., dW[L], db[L]` into one big vector (`d_theta`)
  - Algorithm:

```
eps = 10^-7   # small number
for i in len(theta):
  d_theta_approx[i] = (J(theta1,...,theta[i] + eps) -  J(theta1,...,theta[i] - eps)) / 2*eps
```

  - Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` (`||` - Euclidean vector norm) and check (with eps = 10^-7):
    - if it is < 10^-7 - great, very likely the backpropagation implementation is correct.
    - if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector.
    - • if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation.

Gradient Checking Implementation notes

- Don't use the gradient checking algorithm at training time because it's very slow.
- Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add `lamda/(2m) * sum(W[l])` to `J` if you are using L1 or L2 regularization.
- Gradient checking doesn't work with dropout because J is not consistent.
  - You can first turn off dropout (set `keep_prob = 1.0`), run gradient checking and then turn on dropout again.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

Summary

Initialization summary

- The weights W[l] should be initialized randomly to break symmetry
- It is however okay to initialize the biases b[l] to zeros. Symmetry is still broken so long as W[l] is initialized randomly
- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't initialize to values that are too large
- He initialization works well for networks with ReLU activations.

Regularization summary

1. L2 Regularization

Observations:

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

What is L2-regularization actually doing?:

- L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember:

Implications of L2-regularization on:

- cost computation:
  - A regularization term is added to the cost
- backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices
- weights:
  - weights end up smaller ("weight decay") - are pushed to smaller values.

2. Dropout

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if `keep_prob` is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.