1. bias and variance:

   In deep learning, bias and variance are two crucial concepts that help us understand the behavior of predictive models in terms of their generalization ability. They are part of the bias-variance tradeoff, a fundamental principle that describes the balance between the error introduced due to the model's simplicity (bias) and the error introduced due to its complexity (variance).

Bias

Bias refers to the error due to overly simplistic assumptions in the learning algorithm. High bias can cause the model to miss the relevant relations between features and target outputs (underfitting), meaning the model is not complex enough to capture the underlying structure of the data. In deep learning, a model with high bias might be too shallow, with too few layers or neurons, and thus unable to learn the complexity of the data.

Variance

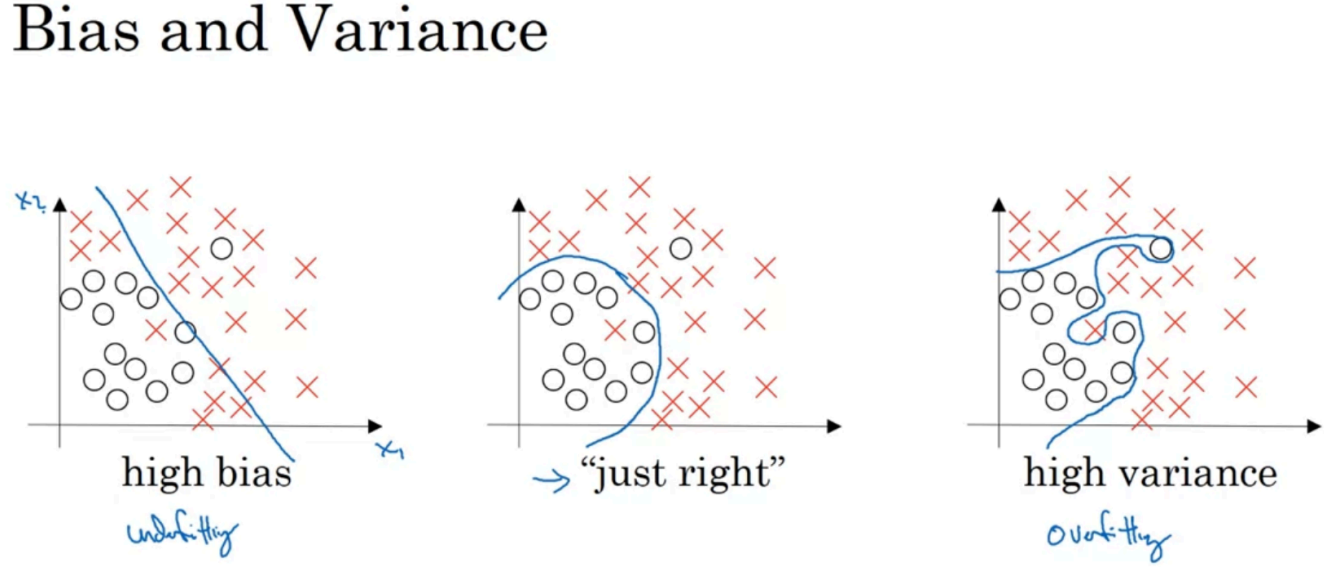
Variance refers to the error due to too much complexity in the learning algorithm. High variance can cause the model to model the random noise in the training data (overfitting), rather than the intended outputs. This means the model is too sensitive to the fluctuations in the training data, and it might not generalize well to unseen data. In deep learning, a model with high variance might have many layers or neurons, making it capable of fitting a wide variety of functions, including noise in the training data.

Bias-Variance Tradeoff

The bias-variance tradeoff is the point where we are adding just enough complexity to the model so that it learns well from the training data, without starting to learn from the noise. The goal is to find a good balance between bias and variance, minimizing the total error. This often involves tuning the architecture of the neural network (such as the number of layers and neurons), as well as regularization techniques to prevent overfitting, and using techniques like cross-validation to estimate model performance on unseen data.

Managing Bias and Variance in Deep Learning

To manage bias and variance in deep learning, practitioners often experiment with the model architecture, add dropout layers or batch normalization, use regularization techniques (like L1/L2 regularization), and ensure they have sufficient data. Using a more complex model can reduce bias but might increase variance; on the other hand, simplifying the model can reduce variance but increase bias. The key is to find a balance that minimizes the total error on unseen data.

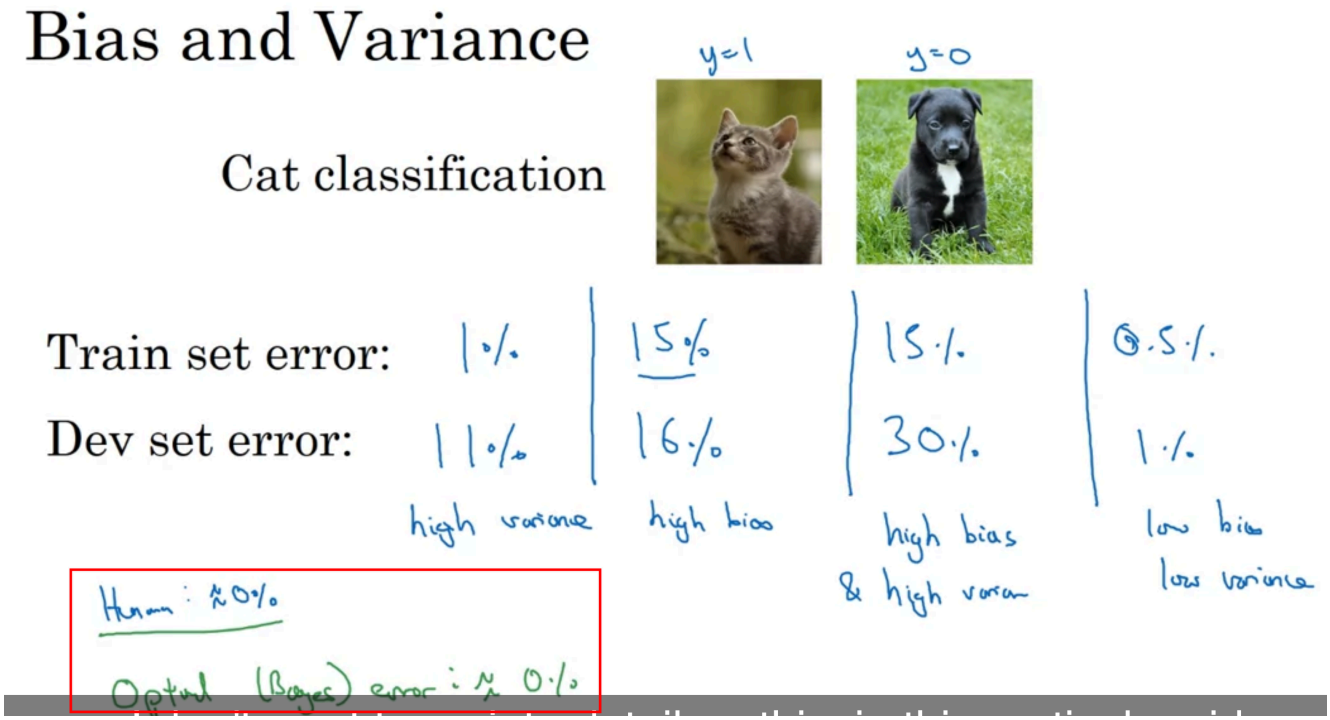


2. Bayes optimal error:

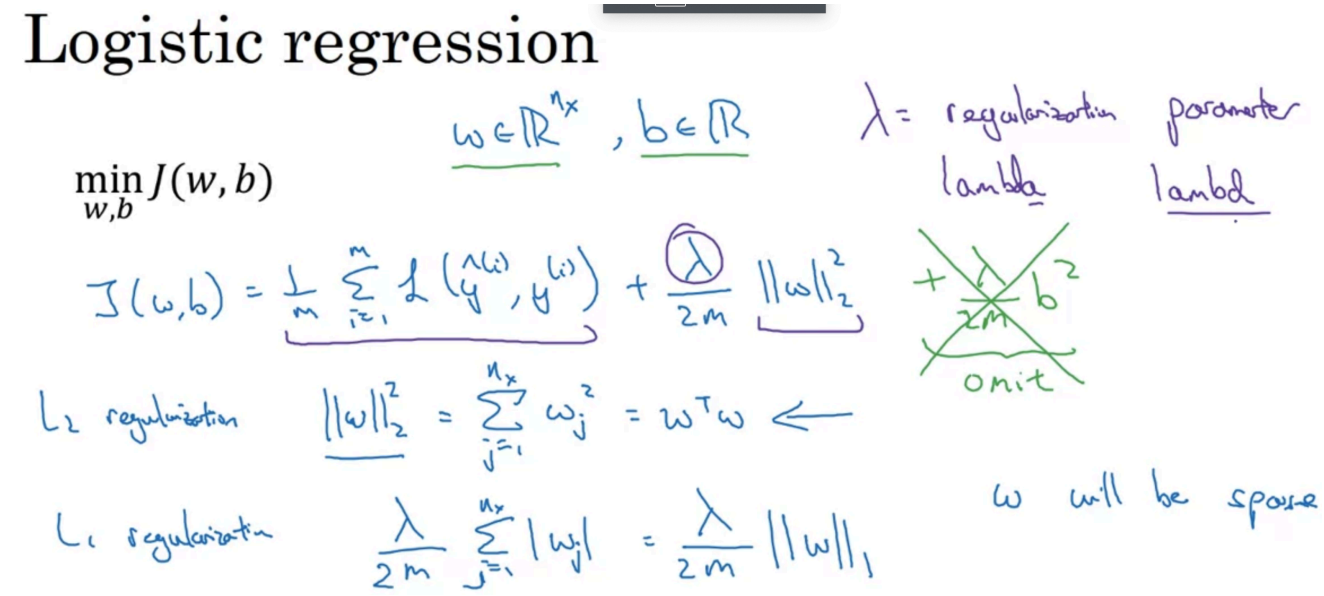
   Bayes optimal error, also known as the Bayes error rate, represents the lowest possible error that any classifier can achieve when predicting the true class. This error rate serves as a theoretical minimum error and is determined by the intrinsic noise within the data distribution itself. It's important to understand that the Bayes optimal error does not stem from the limitations of a particular model or learning algorithm but rather from the complexity and overlap of the underlying data distributions.

   Bias和variance是相对于bayes optimal error来说,如果

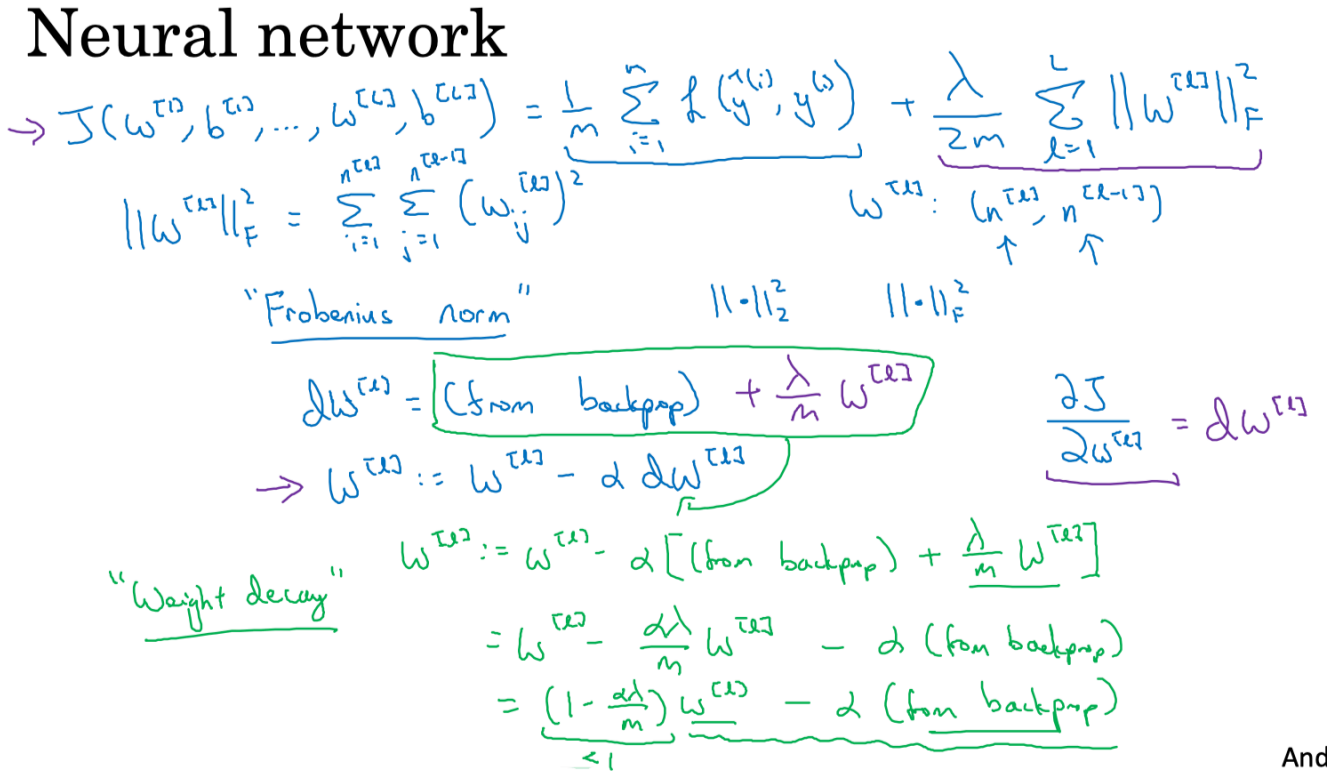
3. Evaluate the bias and variance based on Train and test set error:这是相对于otpimal bayes eror约等于0



4. How to solve Bias and variance?
   1. High bias? 1. Bigger network 2. Training longer 3. Different NN architecture
   2. High variance? 1. More data 2. Regularization 3. Different NN architecture
5. Regularization:

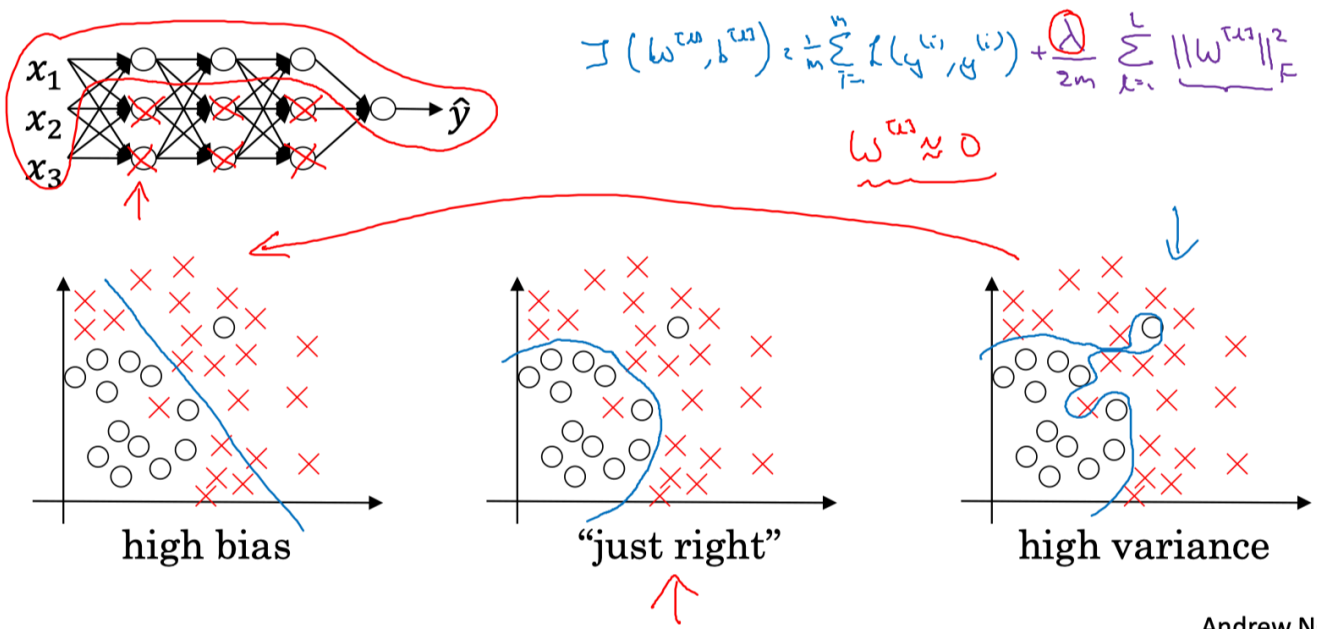


for a neural network: It's called Frobenius norm of matrix instead of L2 norm of matrix here. The L2 regularization is called weight decay 因为最后一行化简后的结果相当于给w乘了个小于1的系数

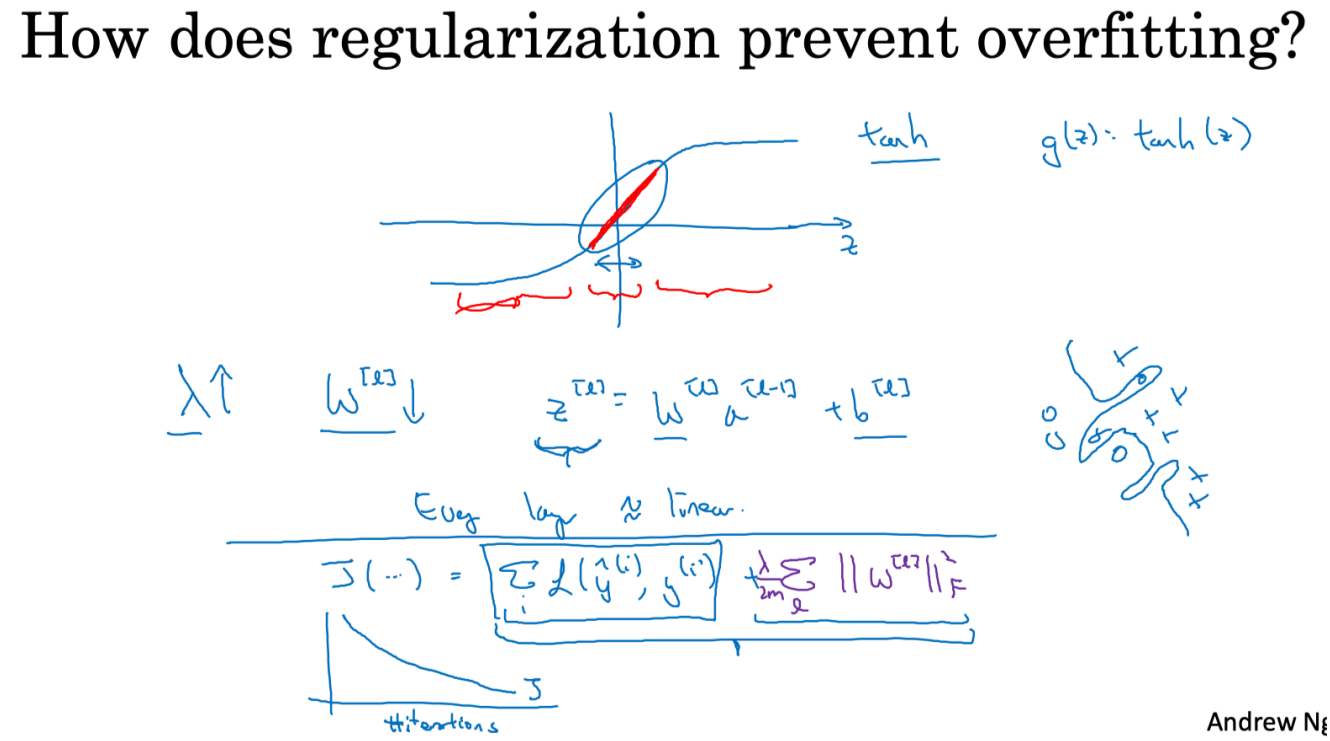


And

6. How does regularization prevent overfitting?



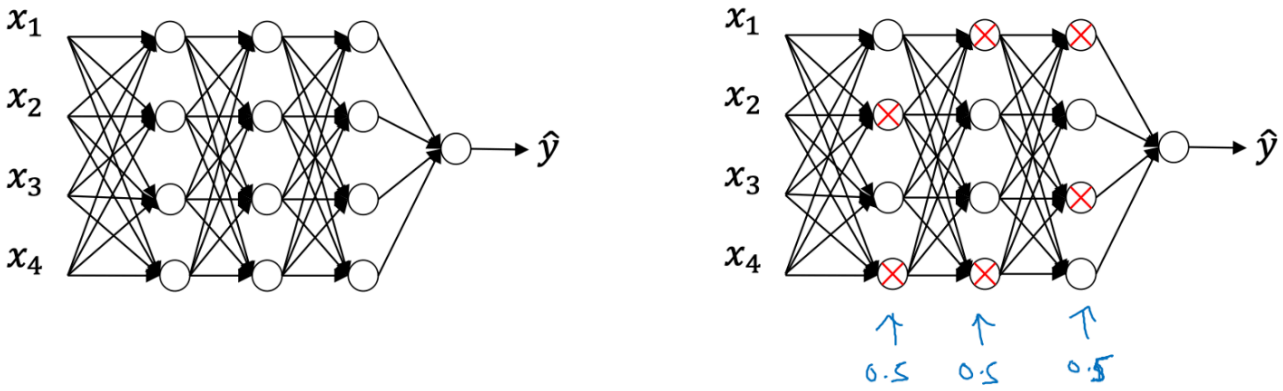
但lambda很大的时候w趋向于0，那么中间有些神经元就约等于被丢掉了就可以防止模型太复杂导致的过拟合big variance问题



还有这种解释就是说通过缩小w将z的范围缩小到红色线性区域，这样就可以将本来过拟合的非线性boundary扭正成线性的

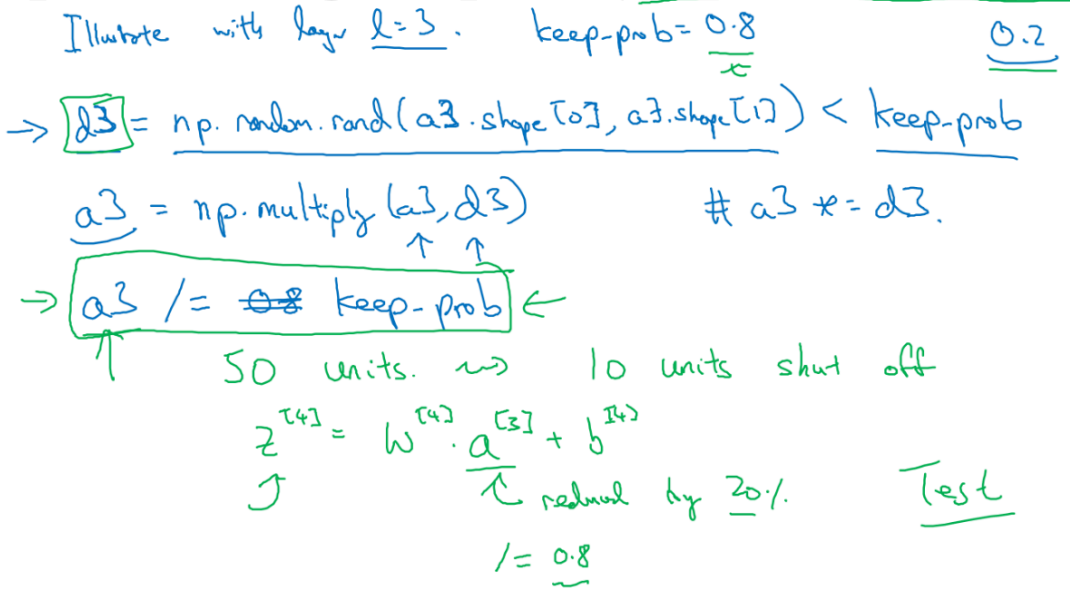
7. Dropout Regularization:

Dropout regularization



Andrew Ng

Implementing dropout ("Inverted dropout")



The image you've uploaded appears to be a handwritten note about implementing dropout in neural networks, specifically a version known as "inverted dropout." Here's how the process described in the image works:

1. `keep_prob = 0.8`: This indicates that during training, each neuron (or unit) in the layer will be kept with a probability of 80%, and dropped with a probability of 20%. This is a common technique used to prevent overfitting in neural networks.
2. `D3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob`: This line of code creates a dropout matrix `D3` the same size as the activation matrix `a3`. `np.random.rand` generates random numbers between 0 and 1. The comparison `< keep_prob` turns these into a boolean matrix where `True` has a value of 1 and represents that the neuron will be kept, and `False` has a value of 0 and means the neuron will be dropped.

3. `a3 = np.multiply(a3, D3)`: This applies the dropout matrix to the activation matrix `a3`. The `np.multiply` function performs element-wise multiplication. Only the activations where `D3` is 1 (True) are kept; the rest are set to 0.
4. `a3 /= keep_prob`: To compensate for the reduced number of active neurons during training, the activations are scaled up by dividing by `keep_prob`. This step is crucial as it maintains the expected value of the activations the same as if dropout was not applied, which is an aspect of "inverted dropout". Without this scaling, the network will tend to perform differently during training and inference (when dropout is not applied).

The illustration also includes a note about the network with 50 units, where ~10 units are shut off, suggesting that this is an example calculation based on the keep probability. The network during testing does not apply dropout, hence the term "test" is written to indicate that this scaling is not performed at test time.

8. why don't do dropout during prediction:

   Dropout is a regularization technique used during the training of neural networks to prevent overfitting. During training, dropout randomly deactivates a subset of neurons (i.e., units in a neural network layer) with a certain probability (like flipping coins to decide which neurons to drop). This forces the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

   However, at prediction (or test) time, dropout is not used, and here's why:

   1. **Removing Randomness**: At prediction time, you want the output of the network to be deterministic. Using dropout during prediction would introduce randomness in the output, which is undesirable. When you make a prediction, you typically want the same output for the same input every time.
   2. **Maintaining Expected Values**: During training, when you use inverted dropout (which involves scaling the activations by the inverse of the dropout probability), you ensure that the expected value of the output remains the same as it would be without dropout. This means that the scaling done during training compensates for the fact that some neurons are dropped. If you were to use dropout at test time without this scaling, the expected values of the activations would be different from what the network learned during training.
   3. **Full Capacity of the Network**: At test time, you want to use the full capacity of the network. The network during training learns to make accurate predictions with a reduced network (because of dropout). So, at test time, when dropout is not applied, the network can use all the neurons, which theoretically could lead to better performance since it's leveraging all available resources.
   4. **Computational Efficiency**: As you mentioned, one could, in theory, run the prediction process many times with dropout and average the results to get an estimate that simulates the training conditions. However, this is computationally expensive and unnecessary because the scaling during training (inverted dropout) already ensures that the activations at test time will be on the correct scale.

   To summarize, during training, dropout helps to simulate a large number of thinned networks with shared weights, which leads to a more robust model. At test time, you want to use the full, learned network to make the most accurate predictions possible. The key is that the scaling during training (due to inverted dropout) ensures that the network's learned weights are appropriate for the full network at test time.

9. Why Dropout work:

   Dropout works as a regularization technique in neural networks for several reasons:

   1. **Prevents Co-Adaptation of Neurons**: By randomly dropping units (neurons) during training, dropout prevents neurons from becoming overly dependent on the specific contributions of other neurons. This means that each neuron must learn to operate well in a variety of different network contexts and cannot rely on any single input too heavily.
   2. **Encourages Redundancy**: Since neurons cannot rely on the presence of others, they must learn more robust features that are useful in conjunction with many different random subsets of other neurons. This encourages the network to create multiple pathways for the information, which can provide redundancy in the representation.
   3. **Effectively Ensembles Many Networks**: During each training iteration, a different "thinned" network is used, which can be seen as sampling from the space of all possible subnetworks of the original architecture. The final model can be thought of as an ensemble of all these thinned networks, which often leads to better generalization.
   4. **Mimics L2 Regularization (to some extent)**: It has been shown that dropout can act as an adaptive form of L2 regularization. The dropout procedure tends to spread out the weights of the neurons, preventing any single weight from having too much influence on the result. This spreading out of weights has an effect similar to L2 regularization, which penalizes the square of the weights.
   5. **Adaptive Regularization**: The regularization effect of dropout is adaptive—it affects each weight differently, depending on the scale of the activations that the weight is multiplied with. This means that the dropout not only regularizes the network but does so in a way that is sensitive to the network's behavior, which is not the case with traditional regularization methods like L2, where every weight is regularized equally.
   6. **Flexibility in Application**: Dropout can be applied differently across layers. For layers where overfitting is a bigger concern (often layers with more parameters), you can set a lower keep probability (higher dropout rate), effectively increasing the regularization. For layers where overfitting is less of a concern, a higher keep probability can be used, or even set to 1.0 (no dropout).
   7. **Computationally Simple**: Dropout is easy to implement and computationally cheap compared to other regularization methods. It doesn't require significant changes to the underlying network architecture or learning algorithm.

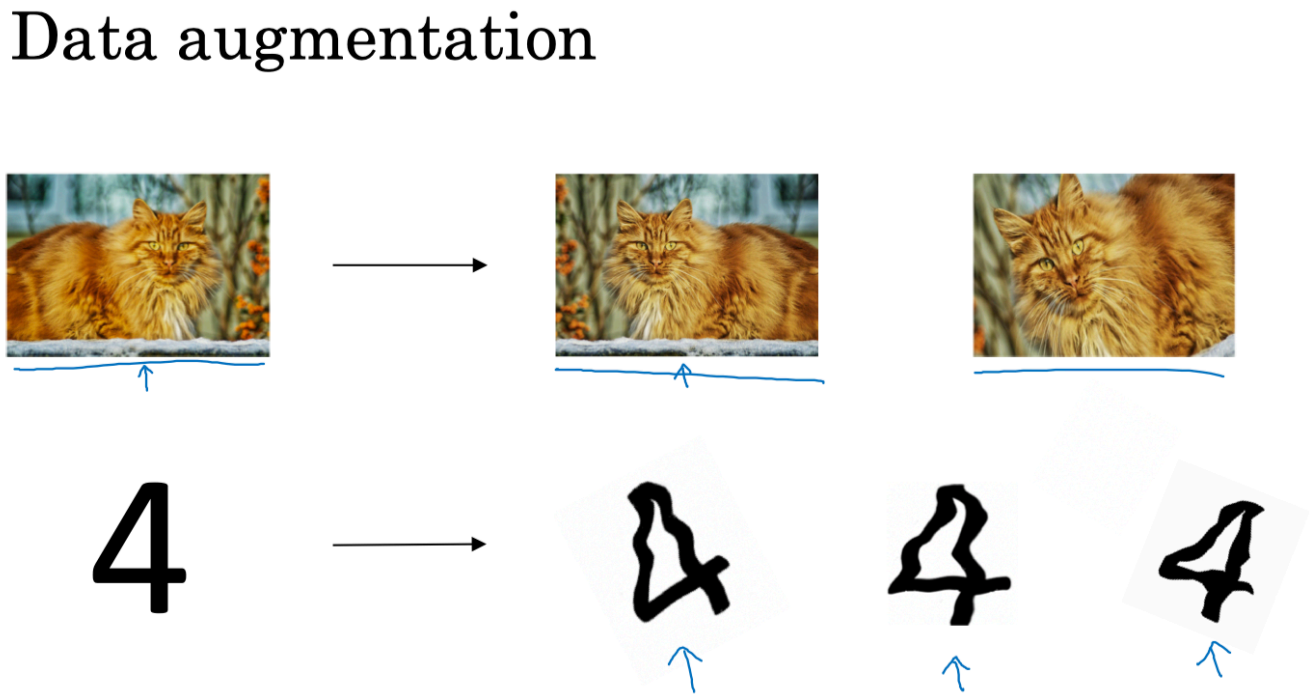
   However, dropout does have some downsides, such as:

   1. **No Well-Defined Cost Function**: With dropout, the cost function is not well-defined because the network architecture is different on each iteration (due to different neurons being dropped). This makes it harder to monitor and ensure the cost function is decreasing over time.
   2. **Increased Complexity in Hyperparameter Tuning**: The use of dropout adds another hyperparameter to tune (the dropout rate), which can increase the complexity of the model selection process.

   In summary, dropout is an effective regularization technique because it encourages the neural network to become less sensitive to the specific weights of neurons, leading to a more generalized model that performs better on unseen data. It does this by introducing randomness during training, forcing neurons to learn to work with different subsets of features, and preventing them from relying

too heavily on any one feature.

10. Other regularization methods:
11. Data augmentation



Data augmentation is a strategy used to increase the diversity of a training dataset without actually collecting new data. This is achieved by applying various transformations to the existing data to create modified versions of the data points. Here's a summary of how it works and its benefits:

Process of Data Augmentation:

1. **Image Transformations**: For image data, common augmentations include flipping the image horizontally or vertically, rotating, zooming in or out, cropping, changing the brightness or contrast, and adding noise.
2. **Audio Transformations**: For audio data, you might add noise, change the pitch, adjust the speed, or apply other filters.
3. **Text Transformations**: For text, augmentations can include synonym replacement, random insertion, swapping words, or deleting words.

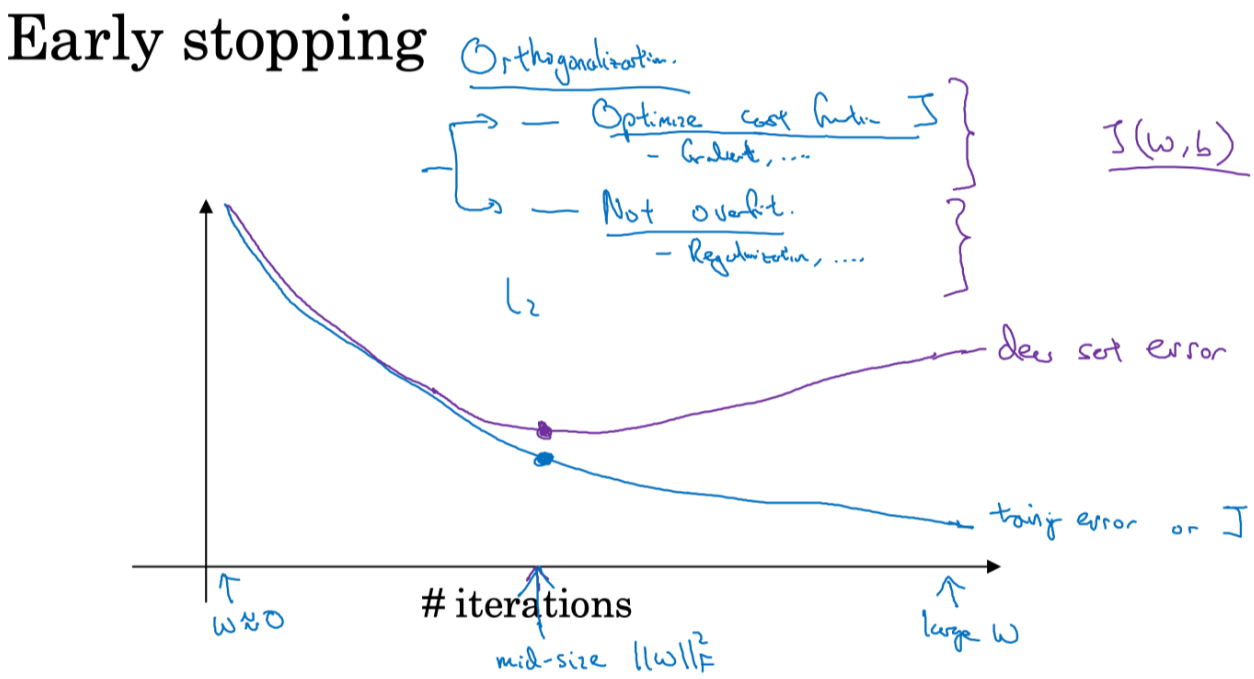
Benefits of Data Augmentation:

1. **Reduces Overfitting**: By creating more varied training examples, the model becomes less likely to learn noise and specific details of the training data, thereby improving its generalization capabilities.
2. **Increases Dataset Size**: Augmentation artificially inflates the dataset size, which can be particularly beneficial when the amount of available data is limited.
3. **Incorporates Invariance**: By teaching the model that certain transformations do not change the underlying class or meaning, the model can learn to be invariant to these transformations. For example, a cat is still a cat whether the image is mirrored or not.
4. **Improves Model Robustness**: Models trained on augmented data can become more robust to variations and noise in real-world data, which they may encounter when deployed.

5. **Cost-Effective**: Data augmentation is a cost-effective way to enhance a dataset since it does not require the time and resources that would be needed to collect and label new data.

In summary, data augmentation is a valuable technique in machine learning for enhancing the size and quality of training datasets, leading to more robust and generalized models. It's particularly useful when the original dataset is small or lacks diversity.

12. Early stopping



(防止test set error上去)

Early stopping is a regularization technique that is used to prevent overfitting in neural networks. Here's a summary of its advantages and downsides:

Advantages of Early Stopping:

1. **Computational Efficiency**: Early stopping saves computational resources since it halts training before the maximum number of iterations is reached.
2. **Prevents Overfitting**: By halting the training process when the validation error starts to increase, early stopping prevents the model from learning the noise in the training data, which can lead to overfitting.
3. **Quick Experimentation**: It allows for faster experimentation with different models because you do not have to train to full convergence each time.
4. **No Need for an Explicit Regularization Hyperparameter**: Unlike methods like L2 regularization that require tuning a hyperparameter (lambda), early stopping inherently provides a form of regularization without the need to search for the best hyperparameter value.

Downsides of Early Stopping:

1. **Coupling of Optimization and Regularization**: Early stopping combines the tasks of fitting the data and regularizing the model, which can complicate the optimization process. It might make it more difficult to discern whether improvements are due to better optimization or better regularization.

2. **Hyperparameter Choice**: While it avoids the need to choose a regularization strength, you still have to decide on other hyperparameters such as when to stop training. The decision of when to stop becomes a hyperparameter itself, which may require validation.
3. **Orthogonalization**: Early stopping violates the principle of orthogonalization, where you ideally have separate tools for optimizing the cost function and for preventing overfitting. This principle makes it easier to understand and control the behavior of the machine learning system.
4. **Potentially Suboptimal Utilization of Data**: By stopping early, you may not fully utilize all the information that the data can provide for the learning process.

To summarize, early stopping can be a useful technique for reducing overfitting, offering a simple way to regularize without extensive hyperparameter tuning. However, it can complicate the training process by combining optimization and regularization into a single step, making it harder to manage the machine learning workflow.

13. Normalization inputs:

$$\hat{X} = \frac{X - \mu}{\sigma}$$

ChatGPT

Normalizing input features is a key technique to accelerate the training process of neural networks. This process involves two main steps:

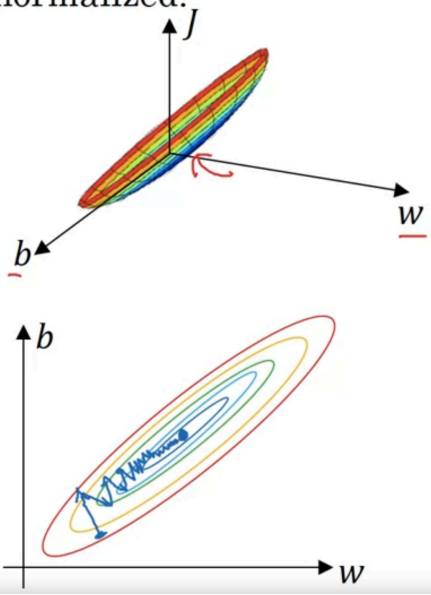
1. **Zero-Centering the Data**: The first step is to adjust the mean of each feature to zero. This is achieved by calculating the mean (μ) of each feature across all training examples and then subtracting this mean from each feature value. Mathematically, the mean μ is calculated as $\mu = \frac{1}{m} \sum_i x_i$, where m is the number of training examples and x_i are the feature values. After computing μ, each feature value is updated as $x := x - \mu$, effectively shifting the data so that its mean is centered at zero.
2. **Normalizing the Variances**: The second step involves scaling the features to have unit variance. This is done by calculating the variance (σ^2) for each feature, which, after zero-centering, is simply the mean of the squared feature values. The variance is calculated as $\sigma^2 = \frac{1}{m} \sum x_i^2$, where x_i are the zero-centered feature values. Each feature value is then divided by the square root of its variance (σ), ensuring that each feature contributes equally to the model, avoiding dominance of features with larger scale.

Normalization is crucial because it addresses the issue of features having different scales, which can lead to a skewed optimization landscape, making the training process inefficient. With unnormalized features, the cost function may become elongated, causing the gradient descent algorithm to oscillate and take smaller steps, thus slowing down convergence. Normalized features, on the other hand, tend to produce a more spherical cost function, allowing for larger gradient descent steps and faster convergence.

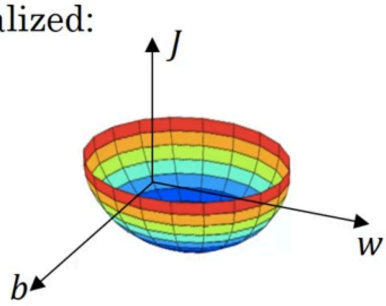
Why normalize inputs?

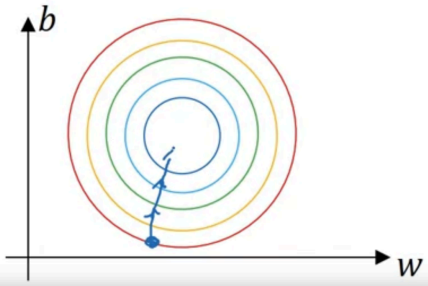
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Unnormalized:



Normalized:

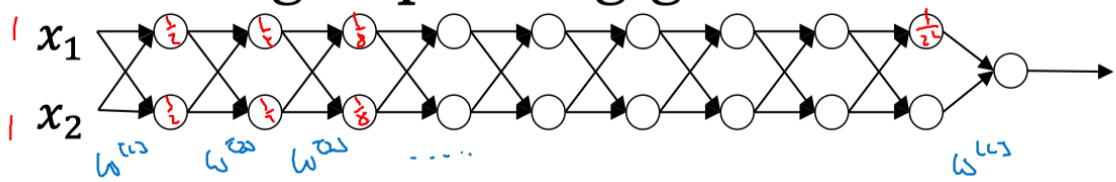




Anc

14. Gradient Exploding or Vanishing:

Vanishing/exploding gradients



在您提供的第一张图片中，表达式

$dz^{[1]} = W^{[2]T}dz^{[2]} * g'^{[1]}(z^{[1]})$

表示在一个两层神经网络中，第一层的误差梯度 $dz^{[1]}$ 是由第二层的误差梯度 $dz^{[2]}$ 通过矩阵 $W^{[2]}$ 的转置 $W^{[2]T}$ 传播回来的，并且与第一层激活函数的导数 $g'^{[1]}(z^{[1]})$ 相乘。

在您提供的第二张图片中，有一个深层网络的例子，在该网络中激活函数 $g(z) = z$，偏置 $b^{[l]} = 0$，而且每一层的权重矩阵 $W^{[l]}$ 相同。在这种情况下，输出 $\hat{y}$ 是通过一系列权重矩阵连乘输入 x 得到的。如果激活函数是恒等函数 $g(z) = z$，那么激活函数的导数 $g'(z)$ 就是1。

让我们将这个情况应用到第一张图片中的公式，假设 $g(z) = z$，那么 $g'(z) = 1$，因此对于所有的层 l，$g'^{[l]}(z^{[l]}) = 1$。这样的话，$dz^{[l]}$ 只会由 $W^{[l+1]T}$ 和 $dz^{[l+1]}$ 决定。由于 $b^{[l]} = 0$，我们可以忽略偏置项对梯度的贡献。

在这种特殊情况下，第 l 层的误差项 $dz^{[l]}$ 将会是：

$dz^{[l]} = W^{[l+1]T}dz^{[l+1]}$

由于 $g'(z) = 1$，这个表达式简化为：

$dz^{[l]} = W^{[l+1]T}dz^{[l+1]}$

这意味着 $dz^{[l]}$ 的大小将直接受到 $W^{[l+1]T}$ 和 $dz^{[l+1]}$ 的影响。如果 $W^{[l]}$ 的每个元素都大于1，那么梯度可能会随着层数的增加而爆炸，因为每次反向传播都会增加梯度的幅度。相反，如果 $W^{[l]}$ 的每个元素都小于1，那么梯度可能会随着层数的增加而消失，因为每次反向传播都会减少梯度的幅度。

在您的例子中，如果权重矩阵 $W^{[l]}$ 的元素都是相同的，并且 $W^{[l]}$ 的每个元素都大于1或小于1，那么梯度将会随着网络深度的增加而爆炸或消失。这是深层网络中梯度爆炸/消失问题的一个直观示例。

梯度爆炸消失产生的原因

梯度消失

梯度消失是指在深度神经网络的训练过程中，从输出层向输入层逐层反向传播时，梯度逐渐变小，直到接近于零。当梯度非常小的时候，权重的更新就会非常微小，几乎不会发生变化，这使得网络的训练停滞不前。

梯度消失的原因主要包括：

1. **深度网络结构**：在深度网络中，梯度是通过反向传播算法从输出层逐层传回输入层的。如果网络很深，梯度在传播过程中会连续乘以小于1的权重，导致梯度指数级减小。
2. **激活函数的选择**：使用某些激活函数（如Sigmoid或Tanh）时，当输入值较大或较小时，这些函数的导数会接近于零。因此，这些激活函数在深度网络中会导致梯度消失问题。

梯度爆炸

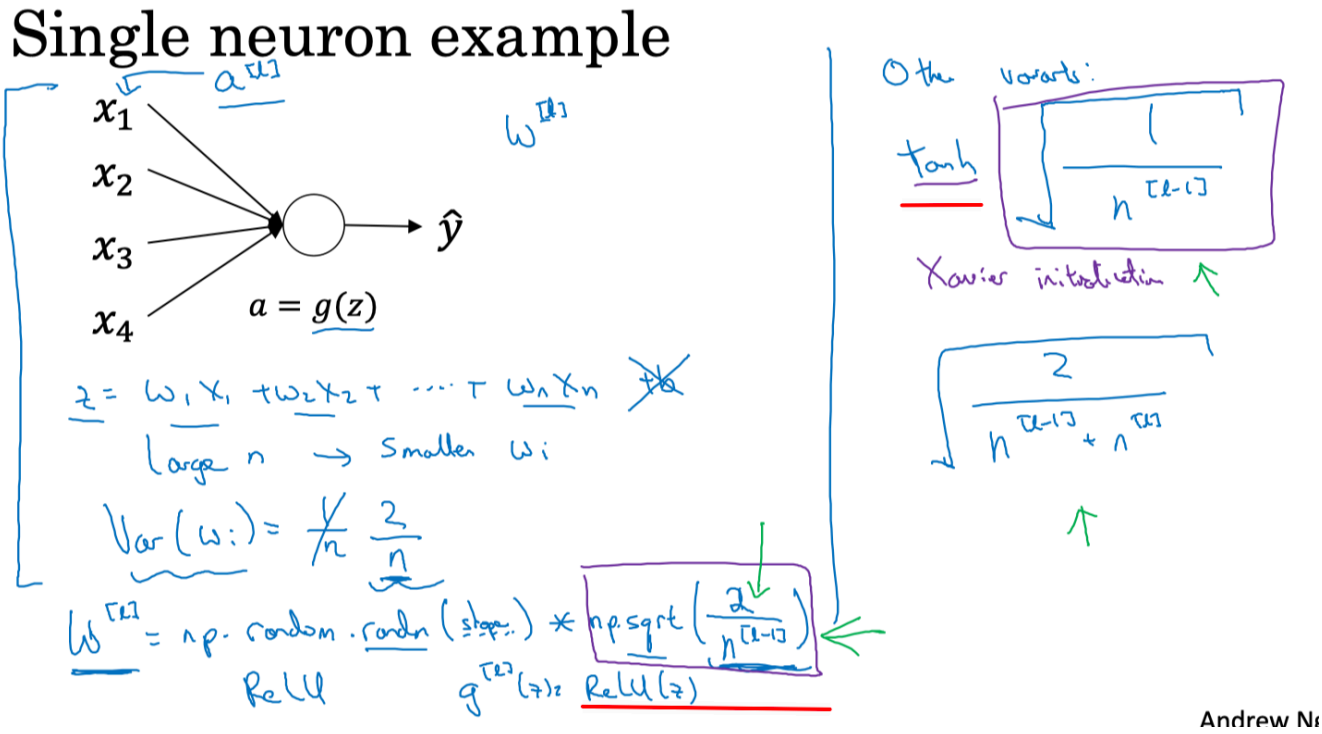
梯度爆炸与梯度消失相反，指的是在深度网络中梯度在反向传播过程中指数级增长，导致权重更新过大。这种过大的更新可以使得模型权重数值变得非常大，以至于网络无法收敛，表现为数值溢出或模型预测结果的剧烈波动。

梯度爆炸的原因主要包括：

1. **深度网络结构**：与梯度消失相似，网络的深度也是梯度爆炸的一个原因。如果梯度在反向传播时连续乘以大于1的权重，梯度就会随着层数的增加而指数级增长。
2. **权重初始化**：权重的初始值过大也可能导致梯度爆炸。较大的初始权重会在反向传播过程中产生较大的梯度，这些梯度累积起来可能会导致爆炸。

15. Weight initialization to solve vanishing/exploding gradient:

- **Weighted Sum** z: In a neural network, each neuron computes a weighted sum of its inputs. This sum z is calculated as $z = w_1x_1 + w_2x_2 + ... + w_nx_n$, where w_i represents the weight and x_i the input feature.
- **Impact of Large Number of Inputs** n: When a neuron has a large number n of input features, the total sum z can become very large if the weights w_i are not properly scaled. This is because each input feature contributes to the sum, and without proper scaling, adding many of these terms can lead to a large z.
- **Desired Property of Weights** w_i: To prevent z from becoming too large or too small, we want each weight w_i to be smaller when the number of inputs n is larger. This helps to balance out the contributions of each term in the sum and keep z within a reasonable range.
- **Variance of Weights** w_i: Setting the variance of the weights to $\frac{1}{n}$ is a strategy to ensure that the weights are scaled down as the number of inputs increases. Variance in this context is a measure of how much the values of w_i are spread out from their mean, which is assumed to be 0 in this case.
- **Initialization of Weight Matrix** W: The practical application of this strategy involves initializing the weight matrix W with random values drawn from a normal distribution (hence the use of `np.random.randn`) and then scaling these values by the square root of $\frac{1}{n}$. This scaling ensures that the weights have the desired variance of $\frac{1}{n}$.



对于relu方差设成2/n, 对于tanh方差设成右上角

16. Gradient checking to debug:

ChatGPT

Gradient checking is a technique used to verify the correctness of the gradients (derivatives) computed by the backpropagation algorithm in neural networks or any other computational graph model. It serves as a debugging tool to ensure that the implemented backpropagation correctly calculates the gradients needed for gradient-based optimization methods, such as stochastic gradient descent. Here's how it works:

1. **Finite Difference Approximation**: The core idea behind gradient checking is to approximate the derivative of a function using the concept of finite differences. Given a function $f(x)$, the derivative $f'(x)$ at a point x can be approximated using the formula:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

   where ϵ is a small value, e.g., $1e-7$ or $1e-4$. This approximation gives us a way to estimate the gradient numerically.
2. **Comparison with Analytical Gradient**: Once we have the numerical approximation of the gradient, we can compare it with the gradient obtained through backpropagation (the analytical gradient). The idea is to compute the numerical gradient for each parameter in the model and compare it with the corresponding analytical gradient computed by backpropagation.
3. **Parameter Perturbation**: To apply gradient checking, we perturb each parameter in the model slightly by ϵ and measure the effect on the output of the function (or the cost function in the context of neural networks). We do this for each parameter individually while keeping others fixed, allowing us to approximate the partial derivative of the function with respect to each parameter.
4. **Threshold for Comparison**: The comparison between the numerical and analytical gradients is usually made using a norm (e.g., Euclidean norm) of the difference between the two gradients. If the norm is below a certain threshold, the analytical gradients are considered to be implemented correctly. The threshold is chosen to be small but accounts for numerical errors inherent in the approximation and floating-point arithmetic.
5. **Limitations and Cautions**: While gradient checking is a powerful tool for verifying the correctness of gradient computations, it is computationally expensive and not practical for use in every iteration of training. It's typically used in a debug mode on a small subset of the model or for sanity checks. Additionally, care must be taken in choosing the value of ϵ; too small, and you may encounter floating-point precision issues; too large, and the approximation may become inaccurate.

Gradient checking is a critical step in the development of neural network models, ensuring that the gradients used for learning are correct and thereby reducing the risk of introducing hard-to-find bugs in the optimization process.

17. Gradient checking implementation notes:

ChatGPT

The video you referenced provides essential insights into gradient checking and its practical implementation within neural networks. Here's a summary of the key points and tips mentioned:

1. **Use Gradient Checking for Debugging Only**: Gradient checking is a computationally expensive process. It's advisable to use it only for debugging purposes to verify the correctness of the gradients computed by backpropagation. During the training phase, rely on backpropagation for gradient computation and disable gradient checking to maintain efficiency.
2. **Diagnosing Failures in Gradient Checking**: If gradient checking fails (i.e., there's a significant discrepancy between the approximated gradients and those calculated by backpropagation), investigate individual components of the gradient vector. This approach helps in pinpointing the source of errors, whether they arise from biases (b) or weights (w), thus guiding you towards the specific part of the code that may contain bugs.
3. **Account for Regularization**: When performing gradient checking, include the regularization term in the cost function if your model uses regularization. This ensures that the gradient computation accounts for the effects of regularization on the cost function, providing a more accurate verification against the approximated gradients.
4. **Handling Dropout**: Gradient checking does not work well with dropout because dropout randomly eliminates subsets of neurons, complicating the computation of a consistent cost function for verification. To circumvent this, disable dropout (set keep-prob = 1.0) when conducting gradient checks to ensure the stability and consistency of the gradients being verified. Once the gradient checking passes, you can re-enable dropout for training.
5. **Gradient Checking at Initialization and After Training**: There's a possibility that backpropagation's accuracy in computing gradients might differ based on the magnitude of weights and biases. Though not commonly practiced, it's suggested to perform gradient checking both at the initial stages of training (when weights and biases are close to zero) and after some training iterations. This dual-phase checking can help in validating the correctness of gradient computations across different stages of model training.
6. **Conclusion and Next Steps**: Gradient checking is a critical tool for ensuring the accuracy of gradient computations in neural networks, especially in the context of debugging. It's part of a broader set of practices aimed at improving model performance, alongside setting up training, development, and test sets, analyzing bias and variance, applying regularization techniques, and implementing strategies for faster training. The video wraps up with encouragement for the viewers to apply these concepts in programming exercises and a look forward to future materials.

These points underscore the importance of gradient checking as a diagnostic tool rather than a step in the regular training process, highlighting specific strategies for its effective implementation in neural network development.