Practical Aspects of Deep Learning

FPT Education

FPT UNIVERSITY

Learning Objective:

- 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 backpropagation implementation

Practical Aspects of Deep Learning

FPT Education

FPT UNIVERSITY

Setting up your ML application

Train/dev/test sets

Applied ML is a highly iterative process

- When training a neural network, you have to make a lot of decisions such as how many layers and hidden units your network should have, the learning rate, and activation functions.
- However, it is almost impossible to correctly guess the right values for all of these on your first attempt.
- Therefore, applied deep learning is an iterative process where you start with an idea and code it up to try it out, and based on the outcome, you refine your ideas and change your choices until you find a better neural network.

Applied ML is a highly iterative process

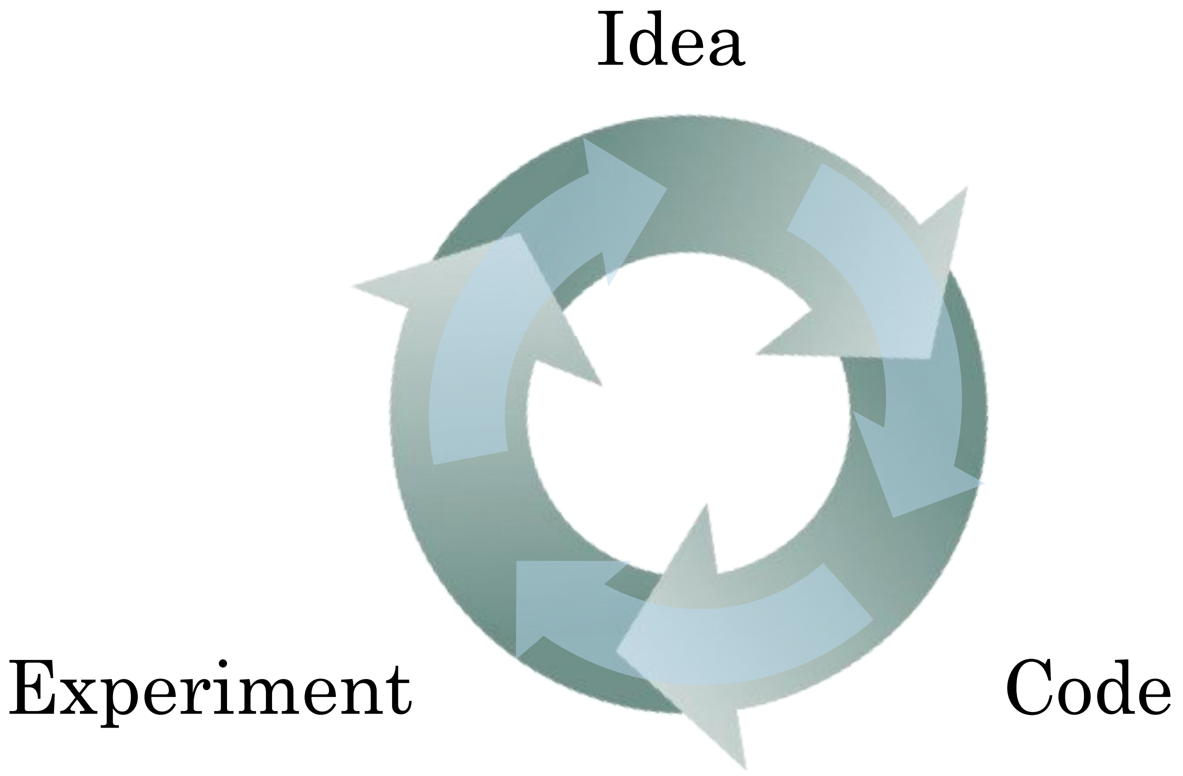
# layers

# hidden units

learning rates

activation functions

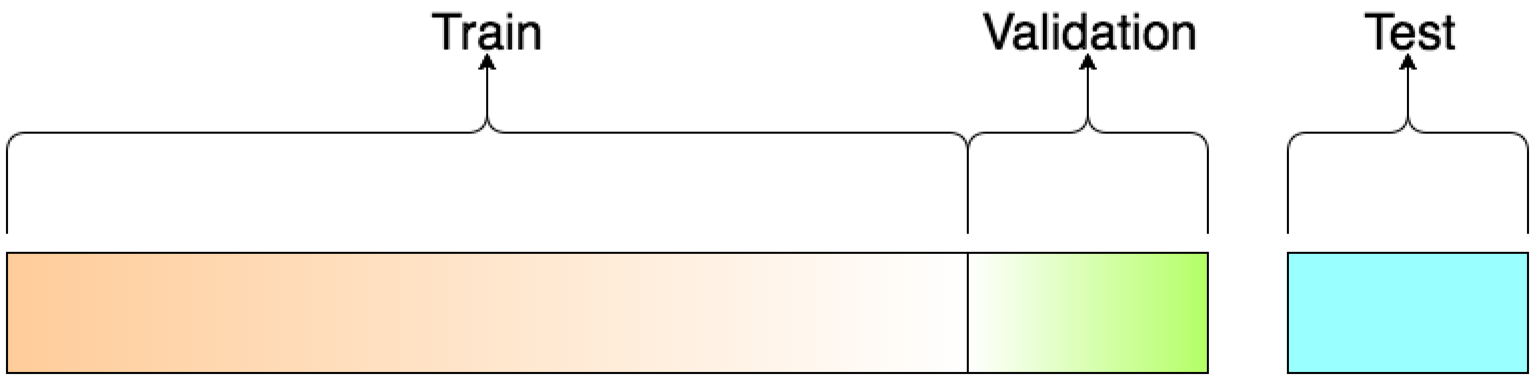
…

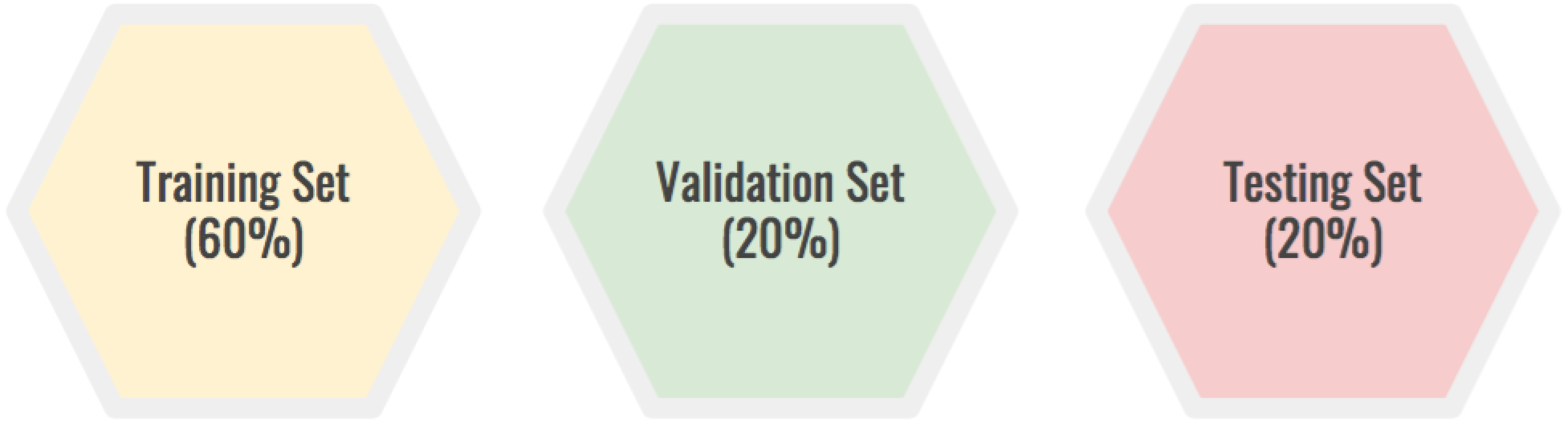


Applied ML is a highly iterative process

- The importance of setting up your data sets well, in terms of your train, development, and test sets, and gives some traditional ratios for splitting the data.
- However, in the modern big data era, it's also fine to set your dev and test sets to be much smaller than your 20% or even 10% of your data. It might be okay to not have a test set if you don't need an unbiased estimate of the performance of your algorithm.
- Setting up a train dev and test set will allow you to integrate more quickly and efficiently measure the bias and variance of your algorithm so you can more efficiently select ways to improve it.

Train/dev/test sets





To train the models

To make sure the models are not overfitting

To determine the accuracy of the models

Mismatched train/test distribution

Training set:
Cat pictures from webpages

Dev/test sets:
Cat pictures from users using your app

Not having a test set might be okay. (Only dev set.)

FPT Education

FPT UNIVERSITY

Setting up your ML application

Bias/Variance

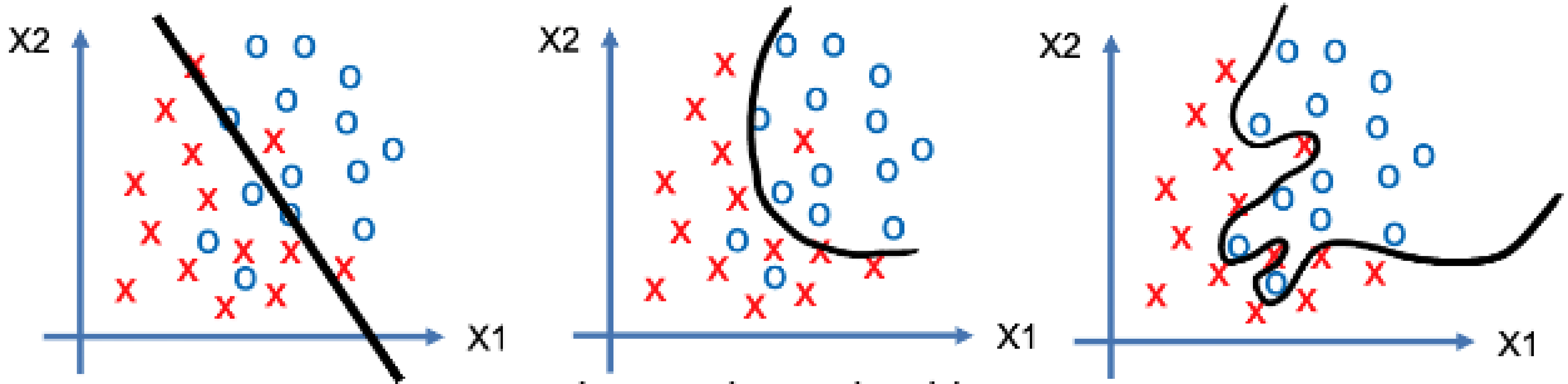
Bias/Variance

- Bias refers to the difference between the expected predictions of a model and the true values of the data.
- Variance refers to the amount by which a model's predictions can vary as new data points are fed into the model.
- A good machine learning practitioner needs to have a sophisticated understanding of this concept because it is easily learned but difficult to master.

Bias/Variance

- In deep learning, there is less discussion of the bias/variance trade-off because there is less of a trade-off in deep learning errors.
- The bias/variance trade-off refers to the balance between underfitting and overfitting.
- Underfitting occurs when the model is too simple and cannot capture the complexity of the data.
- Overfitting occurs when the model is too complex and fits the training data too closely, resulting in poor performance on the test data. The sweet spot is somewhere in between, where the model is just right.

High bias and high variance



High bias
(Under-fitting)

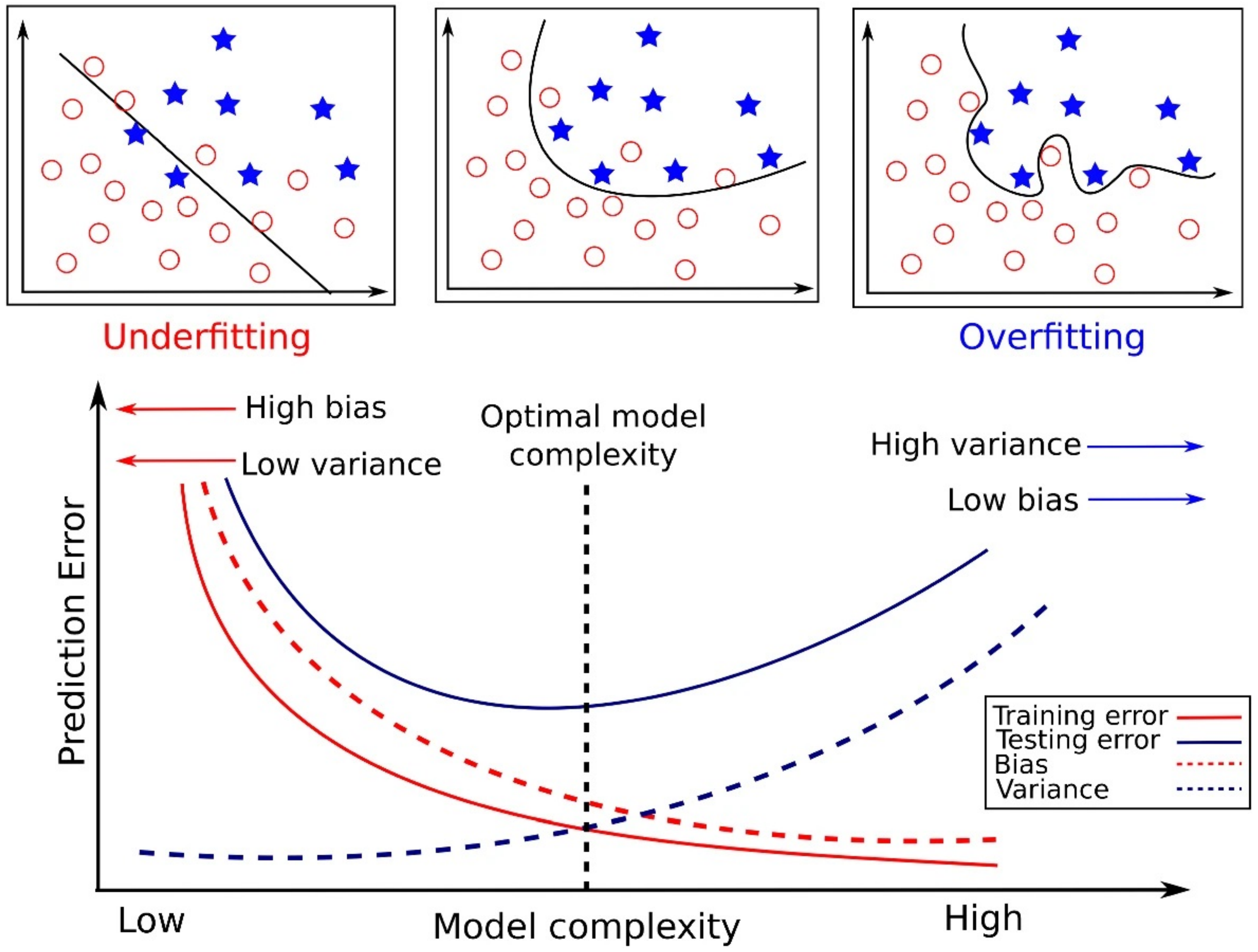
Low variance, low bias
(Just right)

High variance
(overfitting)

Bias/Variance

- To diagnose whether a model has a bias or variance problem, the two key numbers to look at are the train set error and the dev set error:
  - If the training set error is low, but the dev set error is high, then the model may have high variance.
  - If the training set error is high and the dev set error is also high, then the model may have high bias.
  - If the training set error is high, but the dev set error is even higher, then the model may have both high bias and high variance.

Bias and Variance



Bias and Variance

Dog classification

y = 0

y = 1

Train set error:	1 %	14 %	14 %	0.5 %
Dev set error:	11 %	15 %	30 %	1 %
Algorithm diagnosis	high variance	high bias	high bias and high variance	low bias and low variance

Bias/Variance

- The assumption in this analysis is that the Bayes error is quite small, and that the training and dev sets are drawn from the same distribution. If these assumptions are violated, a more sophisticated analysis may be needed.
- By understanding whether a model has high bias or high variance, a machine learning practitioner can try different techniques to improve the model's performance.

FPT Education

FPT UNIVERSITY

Setting up your ML application

Basic “recipe” for machine learning

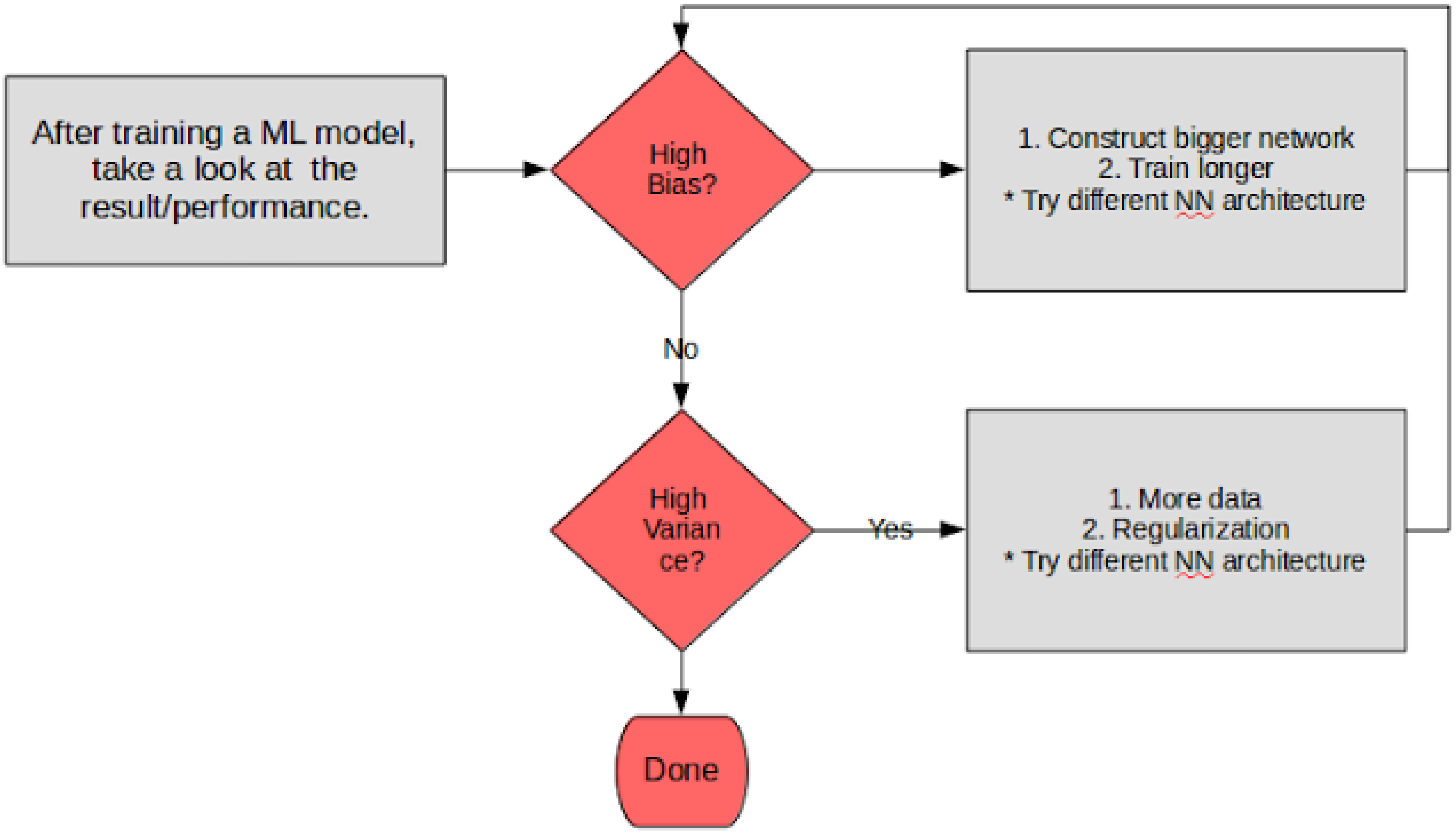
Basic recipe for machine learning

- A basic recipe for machine learning can improve the performance of an algorithm.
- The first step is to diagnose whether the algorithm has a high bias problem.
  - If this is the case, trying to pick a larger network with more hidden layers or hidden units, train for a longer time, or try a more advanced optimization algorithm.
  - If none of these options work, trying a different neural network architecture.
  - The goal is to reduce bias to an acceptable amount.
- The second step is to diagnose whether the algorithm has a high variance problem.
  - To reduce variance, getting more data, using regularization, or trying a more appropriate neural network architecture.
  - The goal is to reduce variance to an acceptable amount.

Basic recipe for machine learning

- The set of things to try depends on whether the algorithm has a high bias or high variance problem.
  - Note: in the modern deep learning era, getting a bigger network almost always reduces bias without necessarily hurting variance, so long as the network is regularized appropriately. The same is true for getting more data, which almost always reduces variance without hurting bias much.
- Finally, regularization is a useful technique for reducing variance but can increase bias slightly.

Basic recipe for machine learning


After training a ML model, take a look at the result/performance.
High Bias?
1. Construct bigger network
2. Train longer
* Try different NN architecture
No
High Varian ce?
Yes
1. More data
2. Regularization
* Try different NN architecture
Done

FPT Education

FPT UNIVERSITY

Regularizing your neural network

Regularization

Regularization

- Regularization helps prevent overfitting in a neural network by adding a penalty term to the cost function that encourages the weights to be small. This smallness of weights can help prevent the network from becoming too complex and overfitting the training data.
- There are several types of regularization techniques which the model tries to minimize during training. The penalty term discourages the model from learning too much from the training data and encourages it to find simpler solutions that generalize better to new data.

Regularization

- One common type of regularization is L2 regularization, also known as Ridge regression, which adds a penalty term to the loss function proportional to the square of the magnitude of the model weights. This encourages the model to learn small weights and reduces the impact of any individual feature on the prediction.

- Another type of regularization is L1 regularization, also known as Lasso regression, which adds a penalty term proportional to the absolute value of the model weights. This has the effect of shrinking some weights to zero, effectively performing feature selection and making the model more interpretable.

Regularization

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2}_{\text{Loss function}} + \lambda \underbrace{\sum_{j=0}^{M} W_j^2}_{\text{Regularization Term}}$$

Regularization

- Finally, there is dropout regularization, which randomly drops out some units in the neural network during training, preventing any single unit from becoming too important and reducing the risk of overfitting.

- Overall, regularization is an important technique for improving the generalization performance of machine learning models, and it should be considered as a standard part of the modeling process.

FPT Education

FPT UNIVERSITY

Regularizing your neural network

Why regularization reduces overfitting

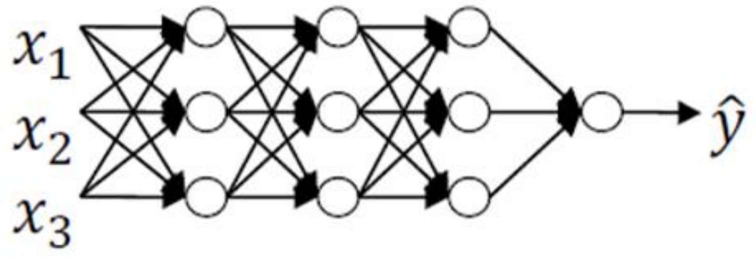
How does regularization prevent overfitting?

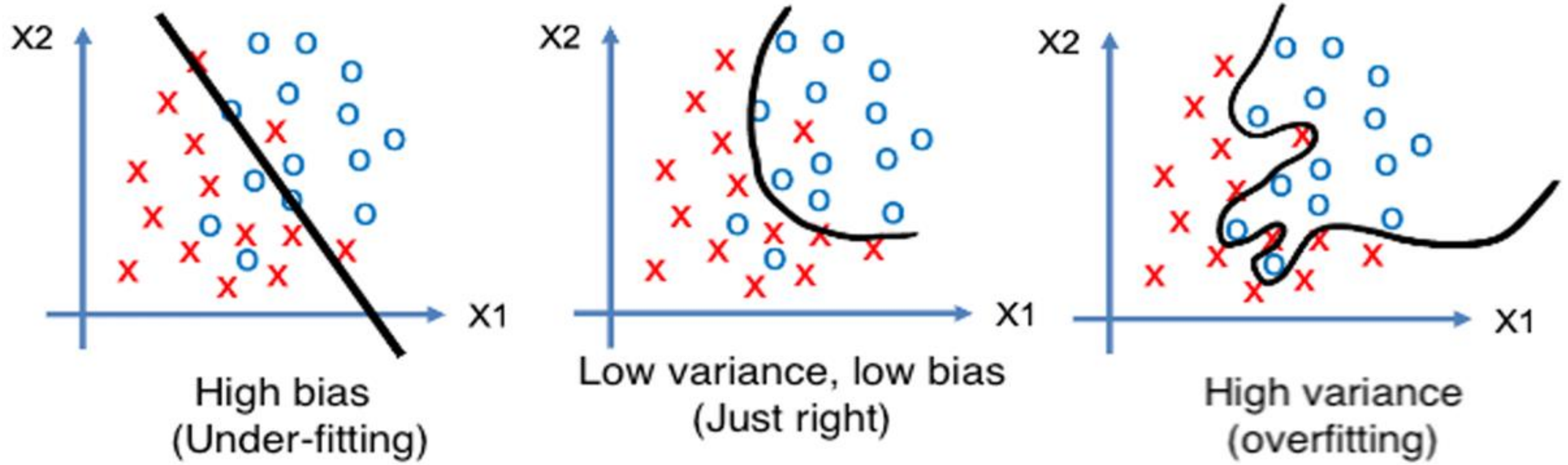
- L2 regularization encourages the model to distribute the weights more evenly across all features rather than relying heavily on a few features. When the regularization parameter λ is set appropriately, it penalizes large weight values, discouraging the model from assigning excessively high weights to any single feature.
- In summary, L2 regularization prevents overfitting by adding a penalty term to the cost function that encourages the model to have smaller and more evenly distributed weights across features. This leads to a more robust and generalizable model, reducing the likelihood of overfitting to the training data.

How does regularization prevent overfitting?

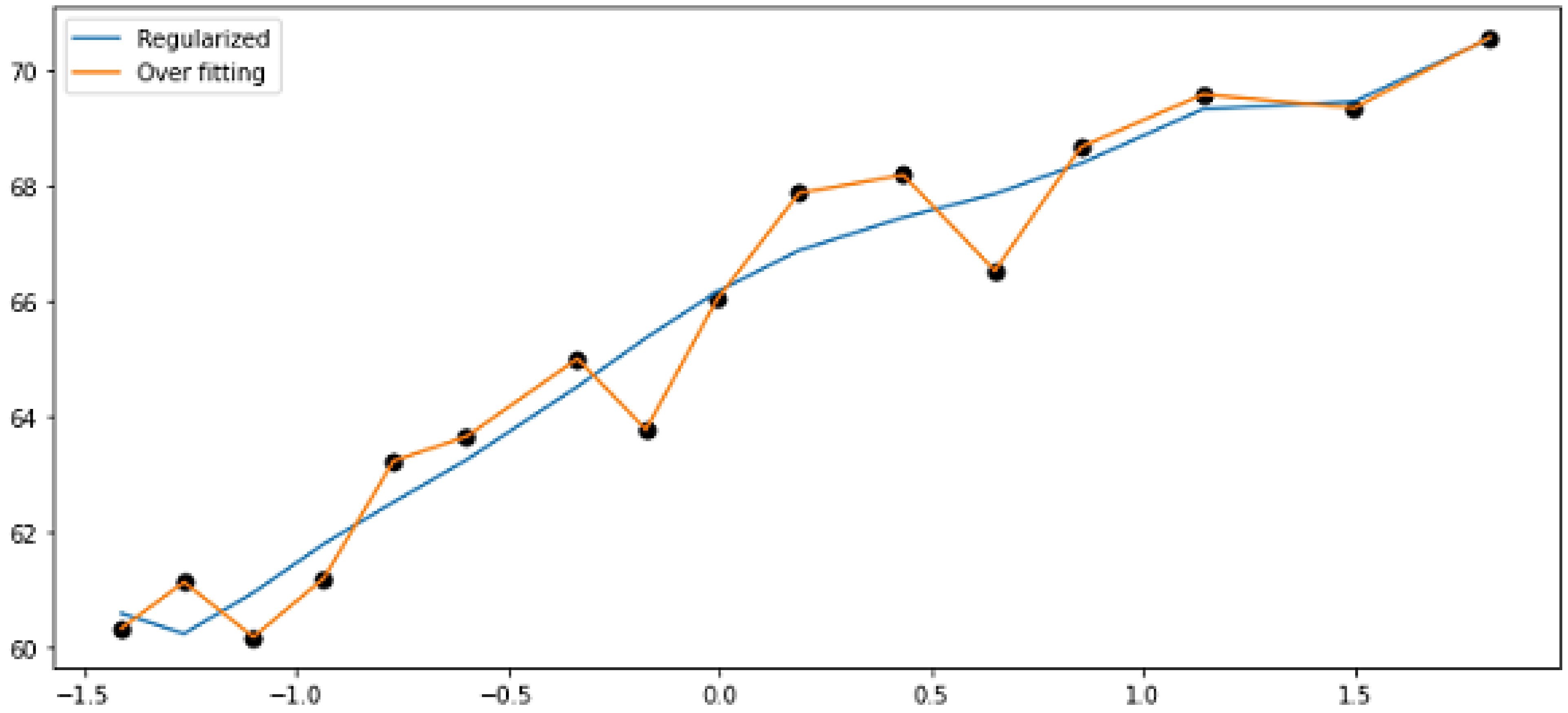
- L1 regularization has the property of introducing sparsity in the model because it encourages many of the model's weights to be exactly zero. When the regularization parameter λ is set appropriately, it effectively forces less important features to have zero weights, effectively excluding them from the model's calculations. In this way, L1 regularization can help the model focus on the most relevant features and reduce the complexity of the learned model.
- In summary, L1 regularization prevents overfitting by adding a penalty term to the cost function that encourages sparsity in the model's weights, leading to a simpler and more generalizable model.

How does regularization prevent overfitting?





How does regularization prevent overfitting?



FPT Education

FPT UNIVERSITY

Regularizing your neural network

Dropout regularization

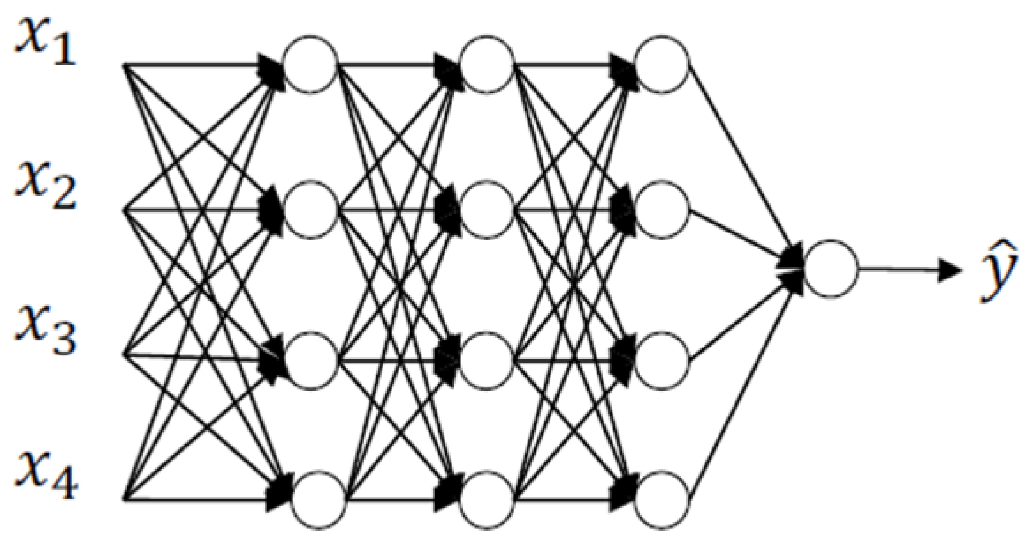
Dropout regularization

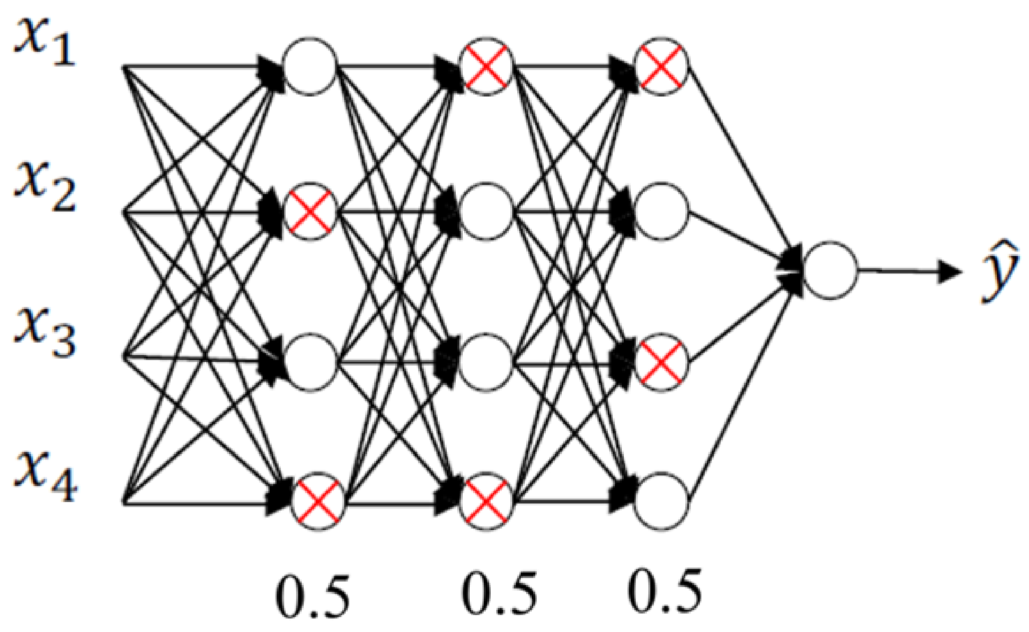
- Dropout is a powerful regularization technique that can help prevent overfitting in neural networks. The basic idea of dropout is to randomly eliminate, or "drop out," some nodes in the neural network during training. This means that the network is effectively training on a smaller, more reduced version of itself.
- To implement dropout, we first set a probability for each node in each layer of the network. For example, we might set a probability of 0.5 for each node in each layer. Then, for each training example, we toss a coin for each node to determine whether or not to keep it. If the coin lands on heads, we keep the node. If the coin lands on tails, we eliminate the node and all its outgoing connections. We then train the neural network using this reduced version of itself.

Dropout regularization

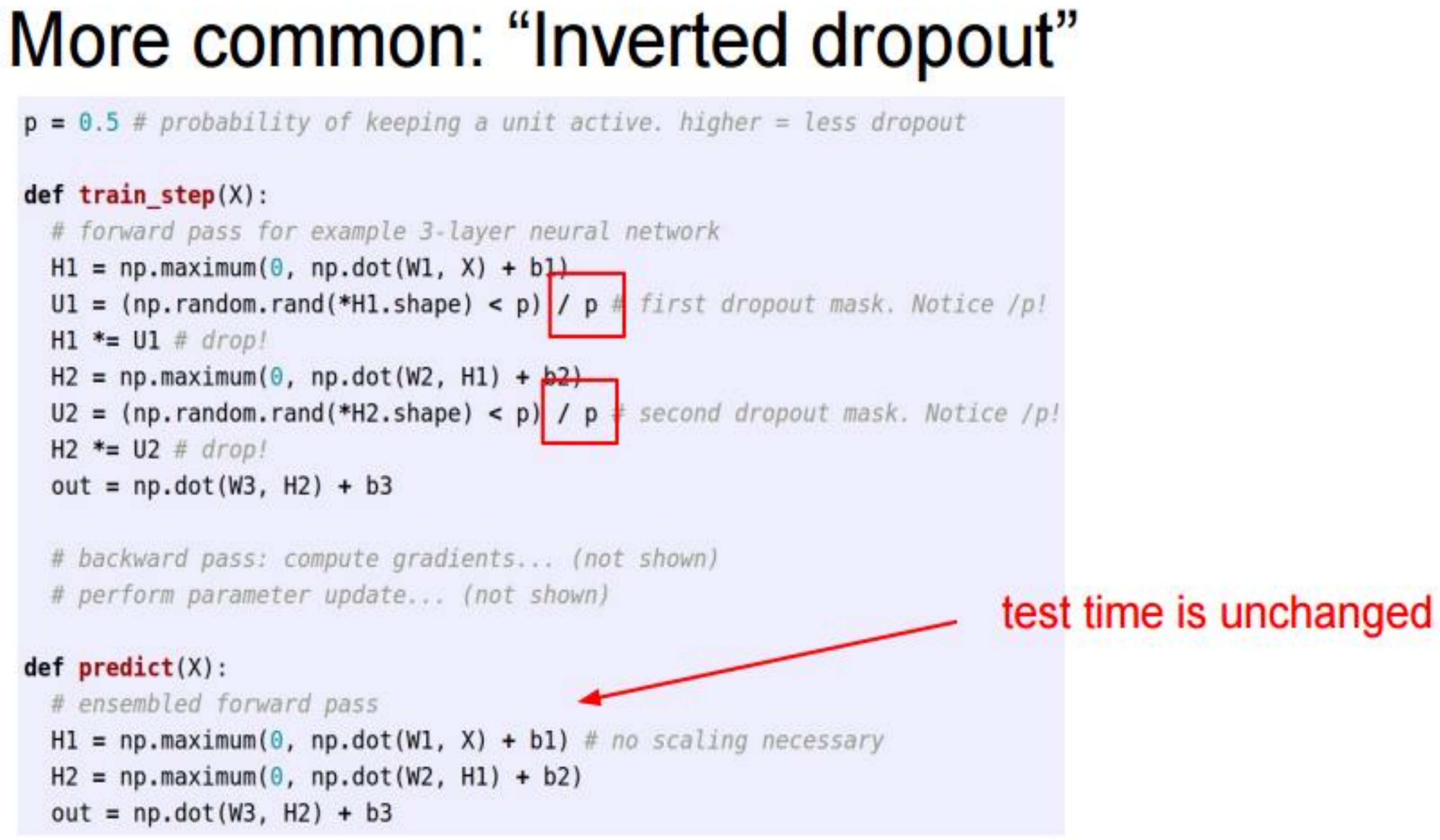
- We implement dropout using a technique called "inverted dropout," which involves scaling up the activations of the remaining nodes by a factor of 1/keep.prob, where keep.prob is the probability of keeping each node. This ensures that the expected value of the activations remains the same.
- At test time, we don't use dropout explicitly, since we don't want our predictions to be random. Instead, we use the full, un-dropped-out neural network to make predictions.
- The reason dropout works is because it helps prevent overfitting by reducing the co-adaptation between neurons. When we drop out some nodes during training, the remaining nodes must learn to perform the task on their own, rather than relying on co-adaptation with other nodes. This can lead to more robust generalization performance on new data.

Dropout regularization





Implementing dropout (“Inverted dropout”)



More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged

Making predictions at test time

- No drop-out

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

FPT Education

FPT UNIVERSITY

Regularizing your neural network

Understanding dropout

Why does drop-out work?

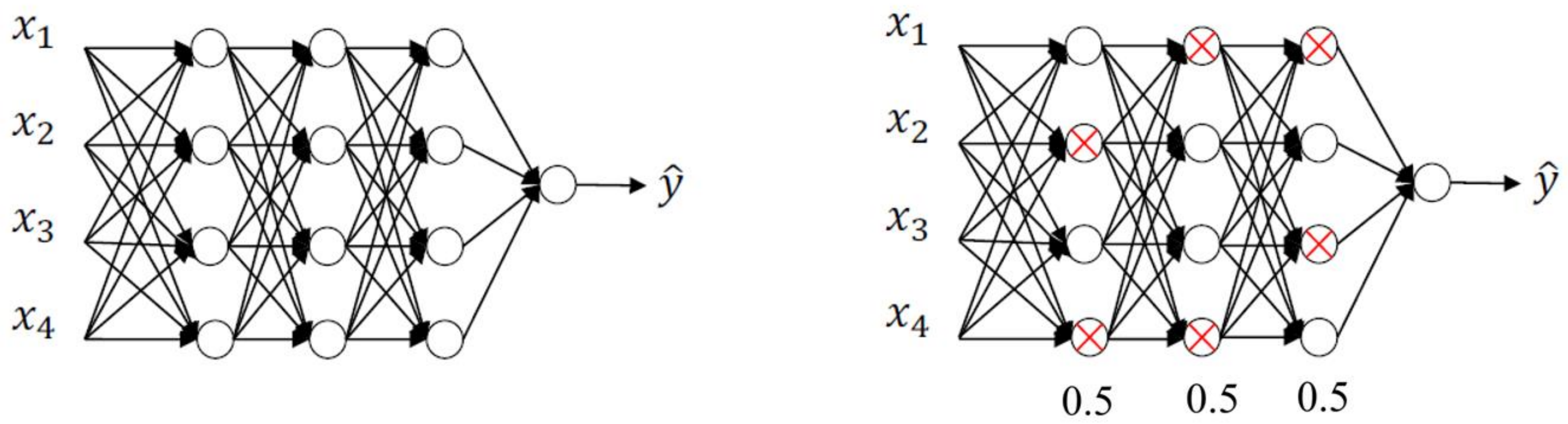
- Dropout encourages weight diversification in the network, preventing overreliance on specific inputs. Random elimination of inputs prompts units to evenly distribute weights, reducing the squared norm, similar to L2 regularization.

- In practice, we select a "keep prop" parameter representing the likelihood of retaining units in each layer. This parameter can be adjusted per layer, with lower values for layers of concern regarding overfitting.

Why does drop-out work?

- Intuition: Can’t rely on any one feature, so have to spread out weights.

Dropout works by **randomly disabling neurons and their corresponding connections**. This prevents the network from relying too much on single neurons and forces all neurons to learn to generalize better. Oct 27, 2021



Why does drop-out work?

- One downside of dropout is that it makes the cost function J less well defined on every iteration, as we are randomly knocking out units. This makes it harder to double check the performance of the network, and we may need to turn off dropout to check that our code is working properly.
- While dropout is frequently used in computer vision, it may be less useful in other application areas where overfitting is less of an issue. There are also other regularization techniques, which we will discuss in the next slides.

FPT Education

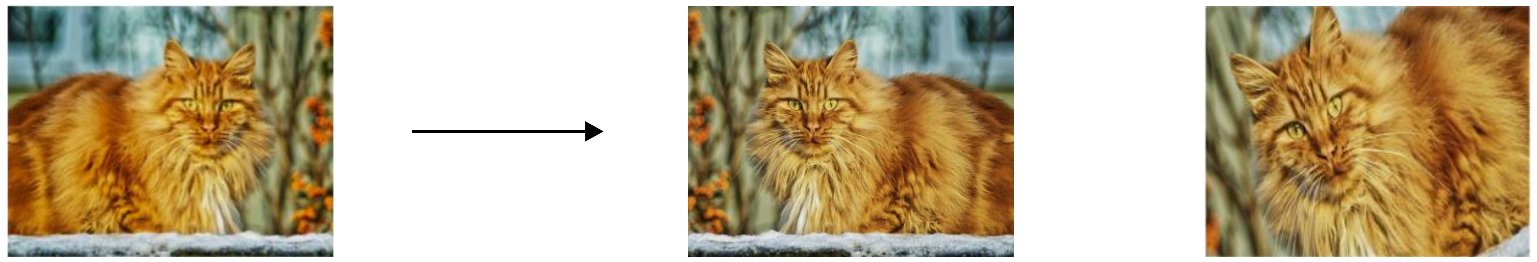
FPT UNIVERSITY

Regularizing your neural network

Other regularization methods

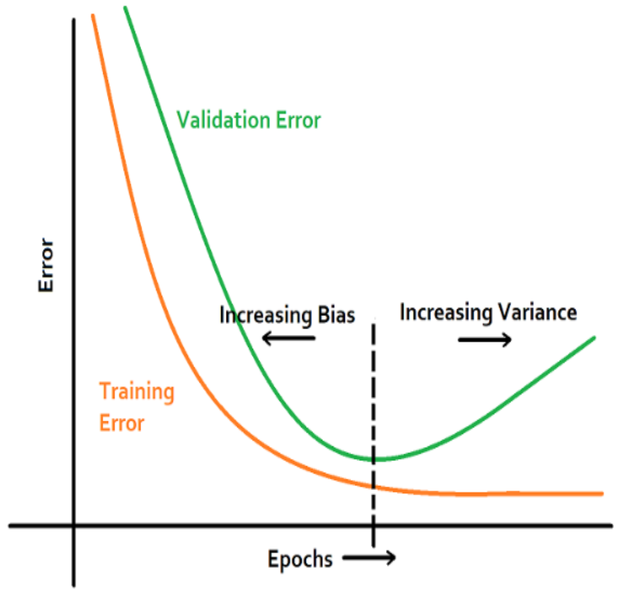
Other regularization methods

- Data augmentation involves creating additional training examples by distorting existing ones, such as flipping images horizontally or taking random crops of an image.
- By doing this, the algorithm is trained on a more diverse set of examples, which can help it generalize better.



Other regularization methods

- Early stopping involves stopping the training process before it reaches the point of overfitting.
- This is done by monitoring the performance of the model on a validation set during training and stopping when the validation error starts to increase.
- By doing this, the model is prevented from continuing to fit the noise in the training data and is forced to generalize better.



Other regularization methods

- However, Early stopping can couple the optimization and regularization tasks, making it more complicated to search for hyperparameters.
- Using L2 regularization instead and trying different values of the regularization parameter lambda, which can make the search space of hyperparameters easier to decompose and search over.

- The concept of orthogonalization involves separating the tasks of optimizing the cost function and reducing overfitting.
- By doing this, it becomes easier to search for hyperparameters and optimize the model efficiently.

FPT Education

FPT UNIVERSITY

Setting up your optimization problem

Normalizing inputs

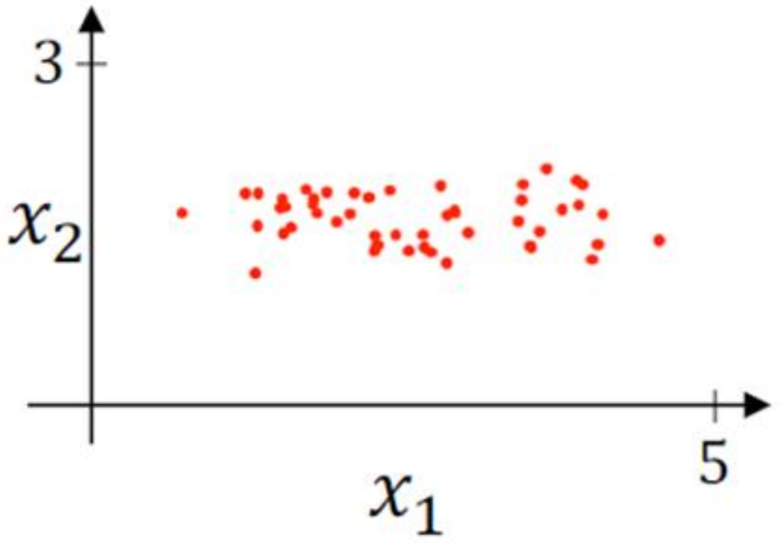
Normalizing training sets

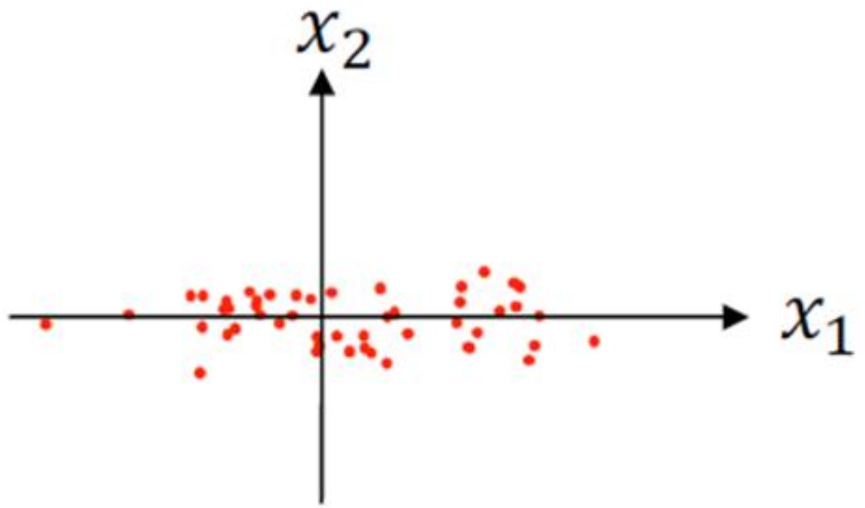
- Normalizing input features means scaling the features so that they are on similar scales. This is done to make the cost function of the neural network more symmetric, which makes it easier and faster to optimize. The normalization process involves two steps:
  - The first step is to subtract the mean from each feature, so that the features have zero mean.
  - The second step is to divide each feature by its standard deviation (variance), so that the features have unit variance.
  - Normalizing input features is important when the input features come from very different scales, as this can make it difficult for the optimization algorithm to converge. If the features are on similar scales, normalization is less important, but it is still a good practice to perform.

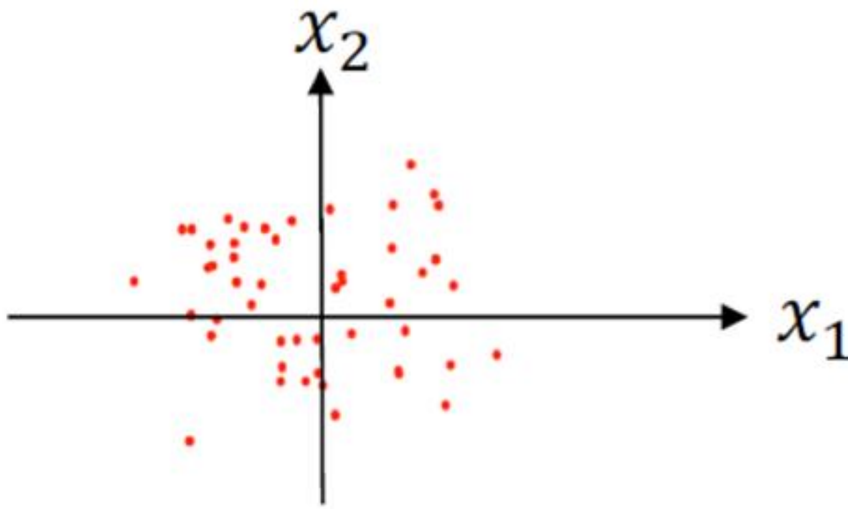
Normalizing training sets

- Note that in the following diagram feature x_1 has very much larger variance than feature x_2 here. After variance normalization, the variance of x_1 and x_2 are both equal to 1.

Normalizing training sets







X = $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Step 1: Mean: μ = $\frac{1}{m}\sum_{i=1}^{m} X^{(i)}$

X = X – μ

Step 2: Variance: σ^2 = $\frac{1}{m}\sum_{i=1}^{m} {X^{(i)}}^2$

X = X/ σ

Normalizing training sets

- It is also important to use the same normalization parameters (mean and standard deviation) for both the training and test sets, so that the test set is scaled in the same way as the training set.
- This ensures that the test set is evaluated on the same features that the neural network was trained on.

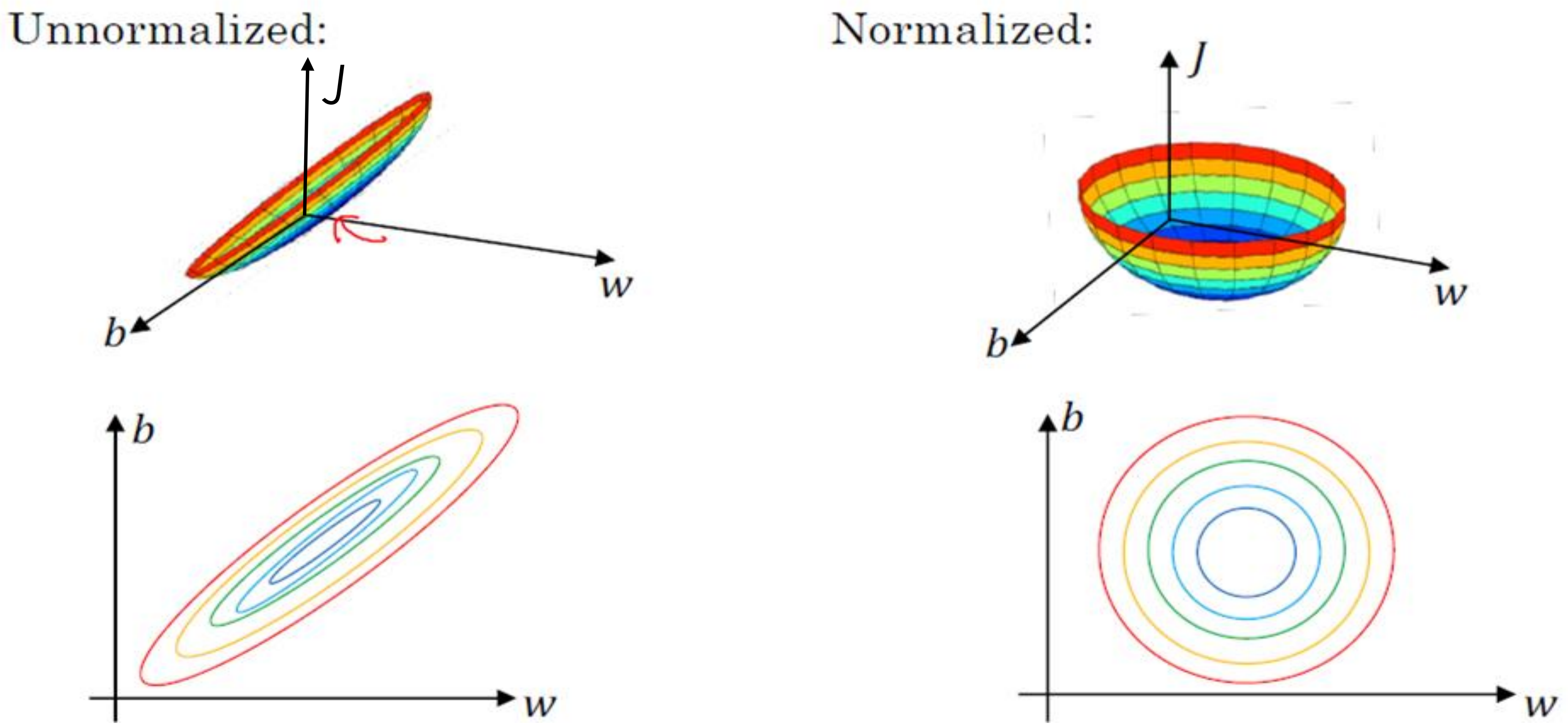
Why normalize input?

- When you training a deep network, the gradient may be very big or very small and this makes training difficult, namely the problem of vanishing or exploding gradient. You can carefully choose the random weight initialization hopefully that your weight will not explode so quickly or decay to zero quickly, so, you can train a reasonable deep network without your weight exploding or vanishing so much.
- For an example: deep network with each neuron having the linear function as the activation function, and set $b^{[i]}$ being zero:
  - If we initialize $W^{[i]}$ a little bigger than the identity matrix, then with very deep networks the activation can explode.
  - If $W^{[i]}$ is just a little bit less than the identity, then the activation will decrease exponentially.

Why normalize input?

- There are several reasons why normalizing input data is important:
- **Improves Convergence**: Many machine learning algorithms, such as gradient-based optimization methods (e.g., gradient descent), converge faster when the input features are on a similar scale. Normalizing the features ensures that all features contribute equally to the learning process, which can speed up the training process and lead to quicker convergence.
- **Prevents Dominance of Features**: If features have different scales, those with larger scales can dominate the learning process. Algorithms might give more weight to features with larger values, even if they are not necessarily more important. Normalizing the features prevents this issue and ensures that no single feature dominates the learning algorithm.
- **Enhances Model Performance**: Normalizing input data can improve the performance and accuracy of the machine learning model. Algorithms like support vector machines, k-nearest neighbors, and neural networks are particularly sensitive to feature scales. Normalizing the features helps these algorithms perform better and make more accurate predictions.
- **Interpretability**: Normalized features make it easier to interpret the model coefficients or feature importance values. When features are on the same scale, it's simpler to compare their importance and understand their contributions to the model's predictions.
- **Regularization**: Regularization techniques (such as L1 and L2 regularization) penalize large coefficients. Normalizing the input features ensures that the regularization term applies uniformly to all features, preventing some features from being unfairly penalized due to their scale.

Why normalize input?



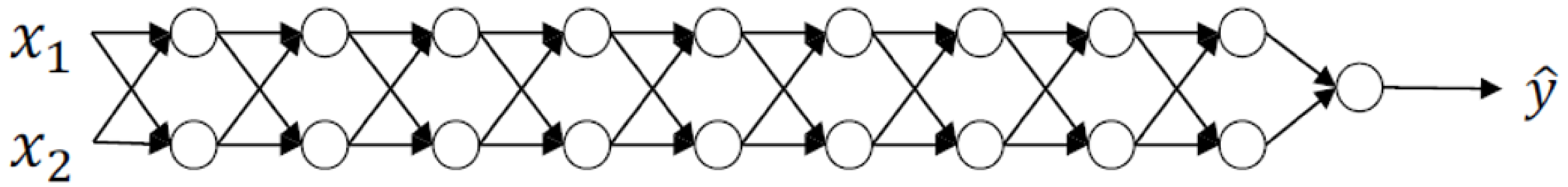
Setting up your optimization problem

Vanishing/exploding gradients

Vanishing/exploding gradients

- In neural network training, the problem of vanishing and exploding gradients can occur, especially in very deep neural networks. This problem arises when the derivatives or slopes become very large or very small:
  - If each weight matrix in a deep network is slightly larger than the identity matrix, then the output can explode exponentially.
  - While if each matrix is slightly smaller than the identity matrix, then the activations can decrease exponentially.
  - Similarly, the derivatives or gradients can increase or decrease exponentially with the number of layers in a deep network.
- This problem makes training difficult, especially if the gradients become exponentially small, as gradient descent will take tiny steps and take a long time to learn anything.

Vanishing/exploding gradients



1. ***Vanishing gradient*** $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 < 1$

2. ***Exploding gradient*** $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 > 1$

Vanishing/exploding gradients

- One solution is to carefully initialize the weights to reduce the impact of vanishing or exploding gradients.
- This can help reduce the problem, but it does not completely solve it.

Setting up your optimization problem

Weight Initialization for Deep Networks

Weight Initialization for Deep Networks

- Zeros initialization: initializing all weights and biases to zero in neural networks, is a simple but discouraged technique. It causes symmetry issues during training, as all neurons in a layer learn the same features, making them redundant and hindering effective learning.
- Random initialization is a common neural network weight initialization method. It breaks symmetry by assigning random values from a chosen distribution to weights and biases. Techniques include Uniform Initialization and Normal (Gaussian) Initialization, chosen based on the activation function.

Weight Initialization for Deep Networks

- ReLU and variants use weight initialization to set the variance proportionate to 1/n, controlling activation and gradient ranges.
- Xavier/Glorot (for linear-like functions) sets variance to 1/n(l-1), where n(l-1) is the number of units feeding into each unit in layer l.
- He initialization (for rectified functions) sets variance to 2/n(l-1), where n(l-1) is the number of units feeding into each unit in layer l.

Weight Initialization for Deep Networks

- He initialization" or "He et al. initialization." It is commonly used for weight initialization in deep neural networks, especially when the activation function is a rectified linear unit (ReLU) or its variants.
- The He initialization formula for the weight matrix of layer l is given as follows:

$W^{[l]}$ = `np.random.randn` * `sqrt(2/n(l-1))`

Where:

$W^{[l]}$: The weight matrix of layer l.

`np.random.randn`: A random number drawn from a standard normal distribution.

`sqrt(2/n(l-1))`: The scaling factor for the weight initialization.

Weight Initialization for Deep Networks

- Weight initialization varies for activation functions:
  - Tanh uses Xavier initialization, setting variance to 1/n instead of 2/n.
  - Another version (Yoshua Bengio and colleagues) uses an alternative formula.
- These formulas provide starting points for weight variance, fine-tuned with hyperparameters.
- Careful initialization aids faster training and mitigates vanishing/exploding gradient issues.

Setting up your optimization problem

Numerical approximation of gradients

Numerical approximation of gradients

- Numerical approximation of gradients is a technique used to estimate the gradients of a function with respect to its parameters by computing finite differences.
- It is commonly used in gradient checking, as described in the Gradient checking, to verify the correctness of analytical gradients computed through methods like backpropagation.

Numerical approximation of gradients

- Given a function f(θ) that depends on a set of parameters θ, the numerical approximation of the gradient of f with respect to each parameter θi can be calculated using the following formula:

  Numerical gradient for θi = (f(θ + ε * ei) - f(θ - ε * ei)) / (2 * ε),

  Where:

  f(θ + ε * ei): The function evaluated at θ with a small perturbation ε added to the ith parameter (θi + ε).

  f(θ - ε * ei): The function evaluated at θ with a small perturbation ε subtracted from the ith parameter (θi - ε).

  ε (epsilon): A small value used to perturb the parameter slightly.

  ei: A unit vector in the direction of the ith parameter (all zeros except at position i).

Numerical approximation of gradients

- The numerical gradient is obtained by calculating the difference between the function evaluations at the perturbed parameter values, divided by 2ε. This finite difference approximation provides an estimate of how the function changes concerning changes in the ith parameter.
- When performing numerical gradient approximation, it is crucial to choose an appropriate value for ε. A value that is too large might lead to inaccurate approximations, while a value that is too small may introduce numerical instability due to finite precision errors. Common choices for ε are in the range of 1e-4 to 1e-8, depending on the scale and nature of the problem.

Numerical approximation of gradients

- Numerical approximation of gradients is a useful tool for verifying the correctness of gradient calculations in complex functions and models.
- However, it can be computationally expensive, especially for high-dimensional models, so it is typically used for debugging and validation purposes rather than in regular training iterations.
- Once the correctness of gradient computations is confirmed, analytical gradients (e.g., obtained through backpropagation) are generally used for efficient parameter updates during the training process.

FPT Education

FPT UNIVERSITY

Setting up your optimization problem

Gradient Checking

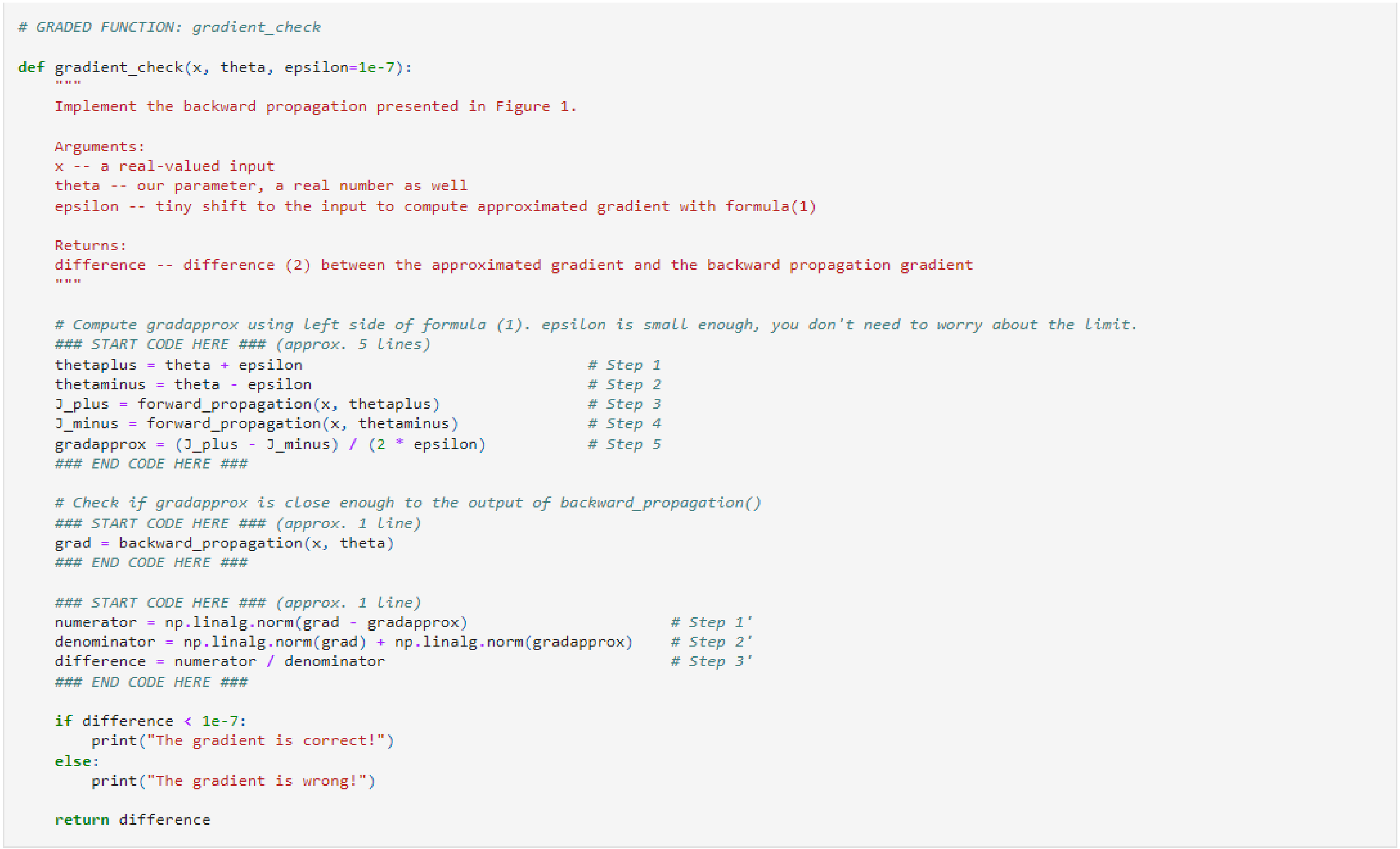
Gradient check for a neuron network

- Gradient checking is a technique used to verify the correctness of the implementation of backpropagation in neural networks.
- Gradient checking steps:
  - Flatten all parameters into a vector "theta."
  - Define cost function J(theta).
  - Compute numerical gradients using finite differences.
  - Use backpropagation to compute analytical gradients.
  - Compare numerical and analytical gradients.
  - Calculate the difference using a suitable metric.
  - Check if the difference is below a predefined threshold to validate the backpropagation implementation.
  - Gradient checking validates and corrects errors in gradient computation, ensuring accurate learning and improved model performance during training with optimization algorithms like gradient descent.

Implement Gradient check

- Setting epsilon to 10-7 in gradient checking ensures the correctness of the derivative approximation. Values much smaller than 10-7 indicate accuracy, while those in the range of 10-5 may need careful inspection. If values are significantly larger than 10-3, potential bugs may exist.
- Gradient checking is a valuable tool for bug detection and boosting confidence in backpropagation implementations.

Gradient checking



```python
# GRADED FUNCTION: gradient_check

def gradient_check(x, theta, epsilon=1e-7):
    """
    Implement the backward propagation presented in Figure 1.

    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well
    epsilon -- tiny shift to the input to compute approximated gradient with formula(1)

    Returns:
    difference -- difference (2) between the approximated gradient and the backward propagation gradient
    """

    # Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need to worry about the limit.
    ### START CODE HERE ### (approx. 5 lines)
    thetaplus = theta + epsilon                               # Step 1
    thetaminus = theta - epsilon                              # Step 2
    J_plus = forward_propagation(x, thetaplus)                # Step 3
    J_minus = forward_propagation(x, thetaminus)              # Step 4
    gradapprox = (J_plus - J_minus) / (2 * epsilon)           # Step 5
    ### END CODE HERE ###

    # Check if gradapprox is close enough to the output of backward_propagation()
    ### START CODE HERE ### (approx. 1 line)
    grad = backward_propagation(x, theta)
    ### END CODE HERE ###

    ### START CODE HERE ### (approx. 1 line)
    numerator = np.linalg.norm(grad - gradapprox)                      # Step 1'
    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)    # Step 2'
    difference = numerator / denominator                               # Step 3'
    ### END CODE HERE ###

    if difference < 1e-7:
        print("The gradient is correct!")
    else:
        print("The gradient is wrong!")

    return difference
```

FPT Education

FPT UNIVERSITY

Setting up your optimization problem

Gradient Checking implementation notes

Gradient checking implementation notes

- Don’t use in training – only to debug
- If algorithm fails grad check, look at components to try to identify bug.
- Remember regularization.
- Doesn’t work with dropout.
- Run at random initialization; perhaps again after some training.

Summarization

- Datasets are split into train (for training), dev (for tuning), and test (for evaluation).
- Bias is underfitting error, and variance is sensitivity to data fluctuations, causing overfitting.
- The machine learning "recipe" involves data gathering, splitting, model selection, initialization, training, validation, and testing.
- Regularization prevents overfitting by adding penalty terms to the cost function, discouraging complexity. It reduces overfitting by penalizing complexity, leading to more generalizable models.
- Dropout regularization randomly drops neurons during training to further reduce overfitting.
- Normalizing inputs involves scaling them to zero mean and unit variance for improved optimization convergence.
- Vanishing/exploding gradients in deep networks occur when gradients become too small or large during backpropagation.

Summarization

- Numerical approximation of gradients using finite differences helps verify the correctness of analytical gradients.
- Gradient checking involves comparing analytical and numerical gradients to ensure accuracy.

Questions

1. If you have 10,000,000 examples, how would you split the train/dev/test set?
2. What criteria should the dev and test sets meet?
3. If your Neural Network model seems to have high variance, which would be promising things to try?
4. Why is it important to set the development and test sets to come from the same distribution?
5. You are training a cat classifier. Your dev set error is much higher than your training set error. What does this imply?
6. What is “data augmentation”?
7. Why should you avoid using data from the test set to make development decisions?
8. What is “early stopping”?
9. What are “Orthogonalization”?