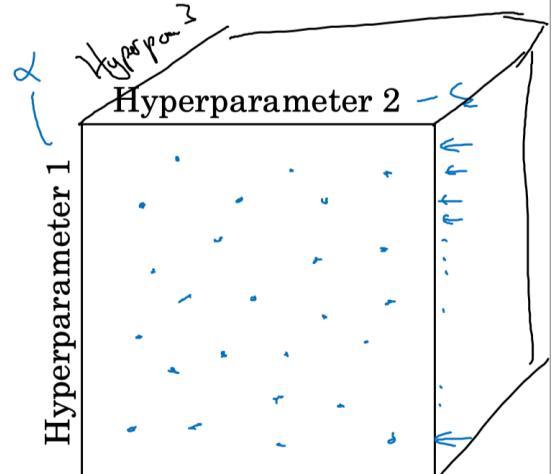
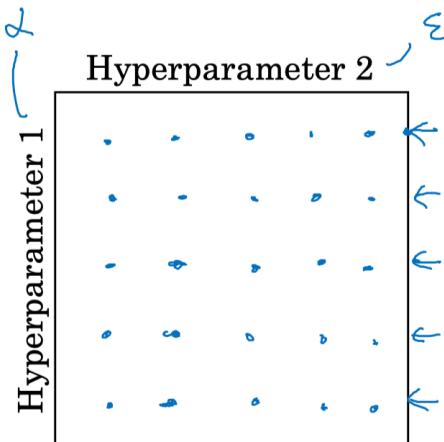


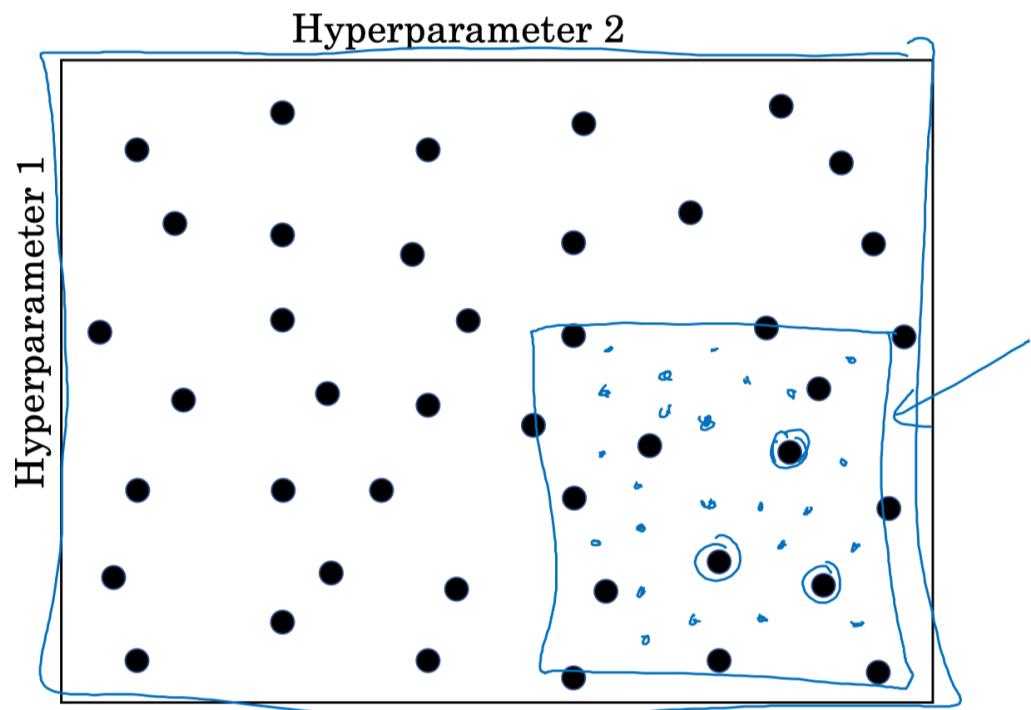
## 1. Hyperparameter Tuning: Try random values don't use a grid:

1. **Importance of Hyperparameters:** Learning rate (alpha) is identified as the most crucial hyperparameter, followed by others like momentum term (beta), mini-batch size, and the number of hidden units. The number of layers and learning rate decay are also important but considered after the aforementioned parameters. For the Adam Optimization Algorithm, default values are often used for beta1, beta2, and epsilon without much tuning.
2. **Random Sampling Over Grid Search:** Unlike traditional machine learning where a grid search might suffice due to fewer hyperparameters, deep learning benefits from random sampling. This approach allows for a broader and more efficient exploration of the hyperparameter space, given the high number of hyperparameters and their varying degrees of impact on model performance.
3. **Coarse to Fine Sampling:** This strategy involves an initial broad random sampling to identify promising regions of the hyperparameter space, followed by more dense sampling in these regions to fine-tune the hyperparameters.

## Try random values: Don't use a grid



## Coarse to fine



## 2. Using an appropriate scale to pick hyper parameters:

1. **Appropriate Scaling for Hyperparameter Exploration:** Random sampling across hyperparameter ranges is more efficient than uniform sampling, but it's crucial to choose the right scale for exploration. Not all hyperparameters should be sampled uniformly over their range.
2. **Examples of Different Scales:**
  - For hyperparameters like the number of hidden units or layers, where a small range of discrete values are considered, uniform sampling might be appropriate.
  - For hyperparameters like the learning rate ( $\alpha$ ), which can vary over several orders of magnitude, logarithmic scaling is more appropriate. This ensures a more balanced exploration across the entire range, preventing overconcentration in a narrow band of values.
3. **Implementation of Logarithmic Scaling:** To sample on a logarithmic scale, generate random values ( $r$ ) within a specified range and then set the hyperparameter (e.g.,  $\alpha$ ) to  $10^r$ .
4. **Special Case - Beta for Exponentially Weighted Averages:** When sampling the beta parameter used in exponentially weighted averages, it's more efficient to sample over  $1 - \beta$  on a log scale. This approach focuses more sampling density where beta is close to 1, which is crucial because small changes in beta near 1 can significantly impact the algorithm's behavior.
5. **Efficient Exploration:** These sampling strategies ensure that the exploration of hyperparameter space is more efficient and effective, particularly for parameters that are sensitive to small changes or span several orders of magnitude.

## Logarithmic Scaling:

To understand logarithmic scaling from a mathematical perspective, let's delve into the core concept and its application, especially in the context of hyperparameter tuning.

### Basic Concept of Logarithmic Scale

A logarithmic scale is based on the logarithm function. The logarithm of a number is the exponent to which the base must be raised to produce that number. Mathematically, if  $b^y = x$ , then  $y = \log_b(x)$ , where:

- $x$  is the number,
- $b$  is the base of the logarithm (commonly 10, e, or 2),
- $y$  is the logarithm of  $x$  to the base  $b$ .

### Application to Hyperparameter Tuning

When applying logarithmic scaling to hyperparameter tuning, you typically deal with a hyperparameter range  $[a, b]$  that spans several orders of magnitude. The goal is to sample values from this range in such a way that each order of magnitude is explored equally.

#### Step 1: Define the Range in Logarithmic Terms

Suppose you want to sample the learning rate  $\alpha$  over a range  $[10^a, 10^b]$ , where  $a < b$ . The values of  $a$  and  $b$  are determined based on the minimum and maximum bounds of the hyperparameter you wish to explore.

#### Step 2: Uniform Sampling in Logarithmic Space

You then uniformly sample a value  $r$  in the linear range  $[a, b]$ . This sampling in the linear range of exponents corresponds to logarithmic sampling in the original hyperparameter space.

#### Step 3: Convert Back to the Original Scale

The sampled value  $r$  is then converted back to the original scale using the inverse of the logarithm function, which is the exponential function, to get the hyperparameter value  $\alpha = 10^r$ .

### Mathematical Formulation

Given:

- A range  $[10^a, 10^b]$  for the hyperparameter.
- A uniformly sampled value  $r$  in the range  $[a, b]$ .

The hyperparameter value  $\alpha$  is calculated as  $\alpha = 10^r$ .

This approach ensures that if  $r$  is uniformly distributed in  $[a, b]$ , then  $\alpha$  is uniformly distributed on a logarithmic scale in  $[10^a, 10^b]$ . This means that the hyperparameter values are not clustered around the higher end of the range but are evenly distributed across each order of magnitude within the range.

### Example

If you want to sample a learning rate  $\alpha$  in the range  $[10^{-4}, 10^0]$  (i.e., from 0.0001 to 1), you would sample  $r$  uniformly between -4 and 0, and then compute  $\alpha = 10^r$ . This method gives equal weight to each decade (0.0001 to 0.001, 0.001 to 0.01, 0.01 to 0.1, and 0.1 to 1), ensuring a balanced exploration of the hyperparameter space.

~ ~ ~

## 3. Hyperparameters Tuning in Practice: Pandas vs. Caviar:

1. **Cross-Fertilization Across Domains:** Ideas and intuitions about hyperparameters can be transferred between different application areas, such as computer vision and speech recognition, leading to innovative applications. However, what works in one domain may not necessarily work in another.
2. **Evolving Hyperparameters:** Hyperparameter settings can become outdated due to changes in the problem, data, or computational environment. It is recommended to reevaluate and possibly adjust hyperparameters periodically, at least every several months.
3. **Hyperparameter Search Strategies:** There are two main approaches to hyperparameter tuning:
  - **The Panda Approach:** Involves "babysitting" a single model, especially useful when computational resources are limited. This method involves gradually adjusting hyperparameters based on the model's performance during training.
  - **The Caviar Approach:** Involves training many models in parallel with different hyperparameters and selecting the best performing model. This strategy is more feasible when ample computational resources are available.

## 4. Normalizing activations in a network: 要么normalize z要么normalize a, 一般normalize z比较多

Batch normalization (BatchNorm) is a technique used to improve the speed, performance, and stability of artificial neural networks. It's applied to individual layers in a network and involves the following steps, typically executed during the training phase:

- Normalization:** For each feature in a mini-batch, BatchNorm normalizes the feature by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Given a feature  $z$  (which is the pre-activation output of a layer, before applying the activation function), the normalized value  $\hat{z}$  is computed as:

$$\hat{z} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Here,  $\mu$  is the mini-batch mean,  $\sigma^2$  is the mini-batch variance, and  $\epsilon$  is a small constant added for numerical stability (to avoid division by zero).

- Scaling and Shifting:** After normalization, BatchNorm introduces two learnable parameters per feature,  $\gamma$  (scale) and  $\beta$  (shift). These parameters are used to scale and shift the normalized values, allowing the network to undo the normalization if that is what the learned behavior dictates. This step ensures that BatchNorm does not restrict the representational capacity of the layer. The scaling and shifting operation is defined as:

$$z' = \gamma \hat{z} + \beta$$

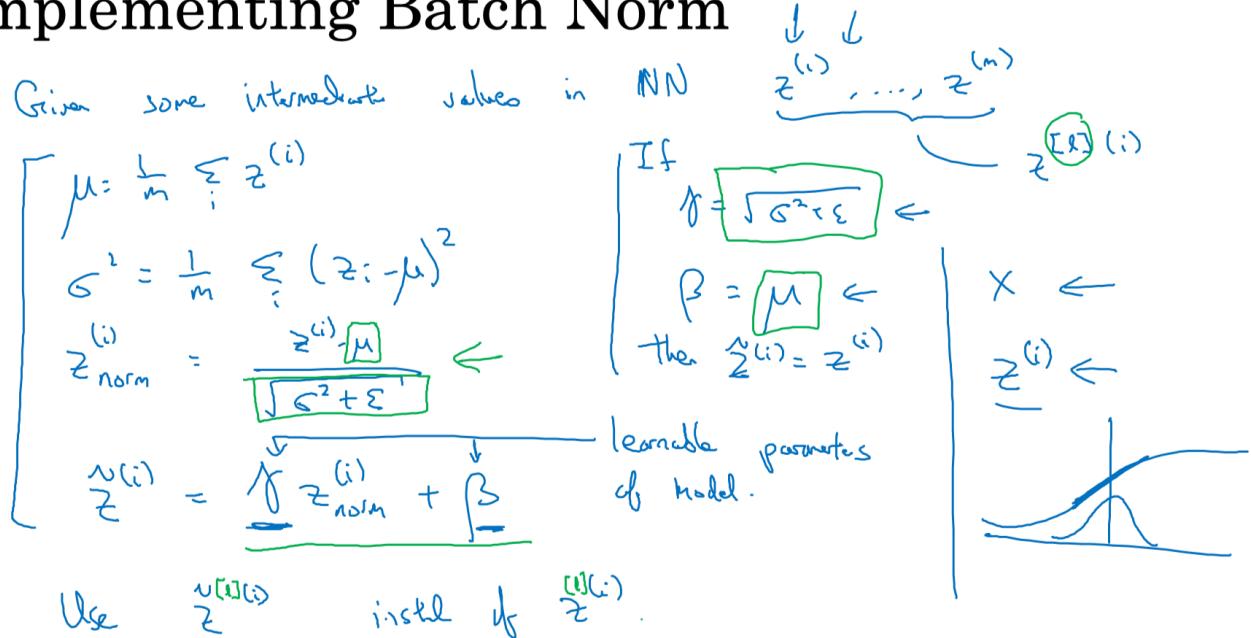
Here,  $z'$  is the output of the BatchNorm layer, which is then passed to the subsequent activation function and layers in the network.

The inclusion of  $\gamma$  and  $\beta$  allows the network to learn an optimal scale and mean for the activations of each layer. Initially,  $\gamma$  could be set to  $\sqrt{\sigma^2 + \epsilon}$  and  $\beta$  to  $\mu$  to maintain the original distribution of  $z$ . However, during training, these parameters are updated through backpropagation along with the other parameters of the network.

BatchNorm's effectiveness comes from its ability to reduce internal covariate shift, which refers to the change in the distribution of network activations due to the updating of weights during training. By normalizing the inputs to each layer, BatchNorm ensures that the optimization landscape is more uniform across different layers, which can lead to faster convergence and increased stability during training. This is particularly beneficial for deep networks, where the distribution of activations can vary greatly across layers, making the network difficult to train.

Batch norm 作用：在神经网络中，每一层的输出成为了下一层的输入，这就形成了一个层层传递的过程。如果在这个过程中，数据的分布发生了变化，也就是所谓的“内部协变量偏移” (internal covariate shift) , 那么模型在学习的时候就会遇到困难。因为每一层都需要不断调整自己，以适应前一层输出数据分布的变化，这就导致了学习过程的不稳定性，特别是在深层网络中，这个问题会更加明显。Batch Norm (批量归一化) 就是为了解决这个问题而提出的。它通过对每一层的输入（也就是前一层的输出）进行归一化处理，确保这些输入数据在经过激活函数之前的分布保持稳定。

## Implementing Batch Norm



The scaling and shifting step in batch normalization, using learnable parameters  $\gamma$  and  $\beta$ , is crucial for several reasons:

- Restoring Representation Power:** Normalization constrains the inputs to a layer to have a fixed mean and variance (usually 0 and 1, respectively). While this can help with training stability and speed, it can also limit the network's ability to represent data since it enforces a specific distribution. By introducing scaling (multiplication by  $\gamma$ ) and shifting (addition of  $\beta$ ), the network can adjust the mean and variance of the activations according to what is learned to be most effective for the task at hand. This step effectively allows the layer to "undo" the normalization if the network learns that the original distribution of activations is more beneficial for learning.
- Maintaining Non-linearity:** Many neural networks rely on the non-linearities introduced by activation functions to learn complex patterns. If the inputs to all activation functions are normalized to a mean of 0 and variance of 1, the activations might predominantly fall into the linear region of functions like the sigmoid or tanh, limiting the network's ability to model non-linear relationships. The scaling and shifting parameters allow the activations to maintain or adjust their non-linearity by controlling the distribution of inputs to the activation functions.
- Flexibility Across Layers:** Different layers in a network might require different statistics for their inputs to optimize performance, depending on the nature of the transformations they perform and their position in the network. The scaling and shifting parameters provide the flexibility needed for each layer to learn the most suitable distribution of its inputs, enhancing the overall capacity and effectiveness of the model.
- Facilitating Learning:** By allowing the network to learn the optimal scale and mean of the activations, the scaling and shifting step can make learning more efficient. It provides an additional degree of freedom, helping the network to converge faster and potentially to a better solution by fine-tuning the distribution of inputs to each layer based on the data and task.

In summary, the inclusion of scaling and shifting in batch normalization is essential for preserving the network's ability to learn complex patterns and relationships in the data, ensuring that the benefits of normalization do not come at the cost of reduced model capacity or expressiveness.

## 5. Fitting Batch Norm into a Neural Network:

1. **Basic Concept:** In a neural network, each neuron (or unit) performs a two-step computation: first, it computes a linear transformation of its input ( $Z$ ) using weights ( $W$ ) and biases ( $B$ ), and then it applies a nonlinear activation function to produce an output ( $A$ ). Batch Norm is introduced between these two steps.
2. **Without Batch Norm:** Normally, the computation in a layer  $l$  would be  $Z_l = W_l A_{l-1} + B_l$ , followed by applying an activation function to  $Z_l$  to get  $A_l$ .
3. **With Batch Norm:** Instead of directly applying the activation function to  $Z_l$ , Batch Norm normalizes  $Z_l$  first. This normalization ( $Z_{\tilde{l}}$ ) adjusts  $Z_l$  to have a mean of 0 and a variance of 1. This is done using the formula:
$$Z_{\tilde{l}} = \gamma_l \left( \frac{Z_l - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \right) + \beta_l$$
where  $\mu_l$  and  $\sigma_l^2$  are the mean and variance of  $Z_l$  across the mini-batch,  $\gamma_l$  and  $\beta_l$  are parameters to be learned, and  $\epsilon$  is a small constant to ensure numerical stability.
4. **Impact on Parameters:** Traditional bias terms ( $B_l$ ) become redundant with Batch Norm, as the normalization step cancels their effect. Thus, the model parameters are updated to include  $\gamma_l$  and  $\beta_l$  for each layer where Batch Norm is applied, alongside the weights ( $W$ ).
5. **Training with Mini-batches:** Batch Norm is typically applied on mini-batches rather than the entire training set. For each mini-batch,  $Z_l$  is computed, normalized to  $Z_{\tilde{l}}$ , and then passed through the activation function to compute  $A_l$ . This process ensures that the inputs to each layer are more standardized, preventing the internal covariate shift and improving training efficiency.
6. **Parameter Update:** During backpropagation, gradients are computed not just for weights but also for  $\gamma_l$  and  $\beta_l$ . The update equations for these parameters are similar to traditional gradient descent:
  - For  $\beta_l$ :  $\beta_l = \beta_l - \alpha \cdot d\beta_l$
  - For  $\gamma_l$ :  $\gamma_l = \gamma_l - \alpha \cdot d\gamma_l$where  $\alpha$  is the learning rate, and  $d\beta_l$  and  $d\gamma_l$  are the gradients of the loss with respect to  $\beta_l$  and  $\gamma_l$ , respectively.
7. **Optimization Algorithms:** Batch Norm can be used with various optimization algorithms such as gradient descent, momentum, RMSprop, and Adam. These algorithms help in updating the parameters ( $W$ ,  $\gamma$ , and  $\beta$ ) based on the computed gradients.
8. **Frameworks and Ease of Use:** Many deep learning frameworks offer built-in functions for Batch Norm, simplifying its implementation to often just a single line of code. This ease of integration helps practitioners focus on model architecture without delving into the low-level details of Batch Norm.
9. **Effectiveness of Batch Norm:** Batch Norm helps in speeding up training and improving the performance of neural networks by addressing the issue of internal covariate shift, i.e., the change in the distribution of network activations due to the update of weights during training. By normalizing these activations, Batch Norm allows for higher learning rates and reduces the sensitivity to initialization.



## No need b in z=wa+b



To explain the redundancy of the bias term when using Batch Normalization (Batch Norm) in neural networks from a mathematical standpoint, let's first define the components involved in these operations:

- Let  $Z^{[l]}$  be the input to a layer  $l$  before applying Batch Norm, where  $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ . Here,  $W^{[l]}$  is the weight matrix,  $A^{[l-1]}$  is the activation from the previous layer, and  $b^{[l]}$  is the bias vector.
- Batch Normalization transforms  $Z^{[l]}$  into  $\hat{Z}^{[l]}$  by applying the following steps:
  1. Compute the mean of  $Z^{[l]}$ :  $\mu = \frac{1}{m} \sum Z^{[l]}$
  2. Compute the variance of  $Z^{[l]}$ :  $\sigma^2 = \frac{1}{m} \sum (Z^{[l]} - \mu)^2$
  3. Normalize  $Z^{[l]}$ :  $Z_{\text{norm}}^{[l]} = \frac{Z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$ , where  $\epsilon$  is a small constant to prevent division by zero.
  4. Scale and shift the normalized value:  $\hat{Z}^{[l]} = \gamma Z_{\text{norm}}^{[l]} + \beta$ , where  $\gamma$  and  $\beta$  are parameters learned during training that allow the network to undo the normalization if that is what the network learns to do.

Now, let's consider the role of the bias  $b^{[l]}$  before batch normalization. If we add a bias  $b^{[l]}$  to  $Z^{[l]}$ , the mean  $\mu$  of  $Z^{[l]} + b^{[l]}$  becomes  $\mu + b^{[l]}$  since the mean of a constant added to a variable is the constant plus the mean of the variable. However, when we normalize  $Z^{[l]}$ , this addition of  $b^{[l]}$  gets subtracted out:

$$Z_{\text{norm}}^{[l]} = \frac{(Z^{[l]} + b^{[l]} - (\mu + b^{[l]}))}{\sqrt{\sigma^2 + \epsilon}} = \frac{Z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

This shows that the addition of  $b^{[l]}$  has no effect on the normalized output, as it gets canceled out during the mean subtraction step. Therefore, the bias term  $b^{[l]}$  becomes redundant when Batch Norm is applied, because its effect is nullified by the normalization process. Consequently, the bias  $b^{[l]}$  can be omitted or considered as permanently set to 0 in layers where Batch Norm is applied, simplifying the model without affecting its ability to learn complex functions.

## 6. Why Batch Normalization works

The key points about why Batch Normalization (Batch Norm) works and its effects on neural network training are as follows:

1. **Normalization Benefit:** Just as normalizing input features to have mean zero and variance one can speed up learning by ensuring all features scale similarly, Batch Norm applies this principle inside the network. It normalizes the activations (hidden unit values) of each layer in the same way. This internal normalization helps in speeding up the learning process by making the optimization landscape smoother.
2. **Covariate Shift Reduction:** Batch Norm makes the network layers less sensitive to changes in the distribution of inputs from previous layers, a phenomenon known as covariate shift. By stabilizing the mean and variance of each layer's inputs, it ensures that the layers are more robust to changes in the network's parameters during training. This stability allows each layer to learn more independently of the others, which can lead to faster convergence and improve the overall training speed.
3. **Slight Regularization Effect:** Batch Norm introduces a slight regularization effect by adding noise to the layer activations. This noise comes from the fact that the mean and variance used for normalization are calculated on mini-batches, which introduces some variability. This effect is similar to dropout in that it prevents the model from relying too much on any single unit, promoting more robust feature learning. However, this regularization effect is relatively mild and not the primary purpose of Batch Norm.
4. **Handling Covariate Shift:** Batch Norm addresses the issue of covariate shift within the network by ensuring that the distribution of the activations doesn't shift too much, even as the network's weights are updated. This makes the learning process for each layer more stable and efficient.
5. **Test Time Consideration:** Since Batch Norm calculates mean and variance statistics on mini-batches during training, a method is needed to handle normalization at test time when predictions might be made on individual samples. Typically, this involves using the moving average of the mean and variance collected during training to normalize test data.

In summary, Batch Norm works by normalizing the activations throughout the network, reducing internal covariate shift, making the network more robust to parameter changes, and speeding up training. It also introduces a slight regularization effect due to the variability in the computed statistics from mini-batches.

## What is Covariate shift:

The key points about why Batch Normalization (Batch Norm) works and its effects on neural network training are as follows:

1. **Normalization Benefit:** Just as normalizing input features to have mean zero and variance one can speed up learning by ensuring all features scale similarly, Batch Norm applies this principle inside the network. It normalizes the activations (hidden unit values) of each layer in the same way. This internal normalization helps in speeding up the learning process by making the optimization landscape smoother.
2. **Covariate Shift Reduction:** Batch Norm makes the network layers less sensitive to changes in the distribution of inputs from previous layers, a phenomenon known as covariate shift. By stabilizing the mean and variance of each layer's inputs, it ensures that the layers are more robust to changes in the network's parameters during training. This stability allows each layer to learn more independently of the others, which can lead to faster convergence and improve the overall training speed.
3. **Slight Regularization Effect:** Batch Norm introduces a slight regularization effect by adding noise to the layer activations. This noise comes from the fact that the mean and variance used for normalization are calculated on mini-batches, which introduces some variability. This effect is similar to dropout in that it prevents the model from relying too much on any single unit, promoting more robust feature learning. However, this regularization effect is relatively mild and not the primary purpose of Batch Norm.
4. **Handling Covariate Shift:** Batch Norm addresses the issue of covariate shift within the network by ensuring that the distribution of the activations doesn't shift too much, even as the network's weights are updated. This makes the learning process for each layer more stable and efficient.
5. **Test Time Consideration:** Since Batch Norm calculates mean and variance statistics on mini-batches during training, a method is needed to handle normalization at test time when predictions might be made on individual samples. Typically, this involves using the moving average of the mean and variance collected during training to normalize test data.

In summary, Batch Norm works by normalizing the activations throughout the network, reducing internal covariate shift, making the network more robust to parameter changes, and speeding up training. It also introduces a slight regularization effect due to the variability in the computed statistics from mini-batches.

## w h y it can reduce covariate shift:



### ChatGPT

Batch Normalization (Batch Norm) helps to prevent covariate shift within the layers of a deep neural network, which can be particularly beneficial for maintaining stable training dynamics. Covariate shift, as mentioned, refers to the change in the distribution of input variables, which can affect the learning process adversely. Here's how Batch Norm addresses this issue:

1. **Normalization of Layer Inputs:** Batch Norm normalizes the inputs to each layer within a network, ensuring that they have a mean of zero and a variance of one (or some other specified mean and variance controlled by learnable parameters). This normalization process is applied to the activations (outputs) of the previous layer before they are used as inputs to the next layer. By doing so, Batch Norm keeps the distributions of inputs to each layer more stable across the training process.
2. **Stabilization of Learning Process:** Without Batch Norm, as the network learns, the weights and biases within each layer change, which can lead to significant shifts in the distribution of layer inputs (activations from the previous layer). These shifts can, in turn, lead to the need for lower learning rates and careful initialization to prevent the gradients from exploding or vanishing. By normalizing the inputs to each layer, Batch Norm reduces the internal covariate shifts, allowing for higher learning rates and less careful initialization, which makes the learning process more stable and often faster.
3. **Maintaining Mean and Variance:** During training, Batch Norm calculates the mean and variance of each layer's inputs based on the current mini-batch and uses these statistics to normalize the inputs. Additionally, it introduces two learnable parameters per activation: one to scale and another to shift the normalized data. This approach ensures that even though the raw inputs to a layer might change as the network's weights evolve, their normalized versions (post Batch Norm) maintain a consistent distribution (e.g., mean zero and variance one), or any distribution that the network learns to be most beneficial via the learnable parameters. This consistency helps in preventing the negative effects of covariate shift within the network.
4. **Learning to Offset Covariate Shift:** The learnable scale and shift parameters in Batch Norm allow the network to undo the normalization if it turns out to be counterproductive for some layers. This means that the network can learn to maintain optimal internal representations for each layer, effectively learning to offset any detrimental effects of internal covariate shift on its own.

In essence, Batch Norm makes the training of deep networks more robust to the internal covariate shift by ensuring that the distribution of inputs to each layer remains more stable throughout training. This stabilization allows each layer to learn on a more consistent "foundation," which can lead to improved training efficiency and model performance.

## 7. Batch Norm at test time:

During training, for a given layer and a mini-batch of size  $m$ , Batch Norm computes the mean ( $\mu_B$ ) and variance ( $\sigma_B^2$ ) for that mini-batch:

1. **Mean Calculation:**

$$\mu_B = \frac{1}{m} \sum_{i=1}^m z^{(i)}$$

Where  $z^{(i)}$  is the input to Batch Norm for the  $i^{th}$  example in the mini-batch.

2. **Variance Calculation:**

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (z^{(i)} - \mu_B)^2$$

3. **Normalization:**

$$\hat{z}^{(i)} = \frac{z^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$\epsilon$  is a small constant added for numerical stability.

4. **Scaling and Shifting:**

$$z_{\text{norm}}^{(i)} = \gamma \hat{z}^{(i)} + \beta$$

Where  $\gamma$  and  $\beta$  are learnable parameters of the Batch Norm layer.

At test time, we cannot use the mean and variance of a single example or a smaller batch because they may not represent the distribution of the dataset. Instead, we use estimates of  $\mu$  and  $\sigma^2$  derived during training. These are typically calculated as exponentially weighted averages across mini-batches:

• **Exponentially Weighted Average for Mean:**

$$\mu = \alpha \mu_{\text{prev}} + (1 - \alpha) \mu_B$$

• **Exponentially Weighted Average for Variance:**

$$\sigma^2 = \alpha \sigma_{\text{prev}}^2 + (1 - \alpha) \sigma_B^2$$

Where  $\alpha$  is a momentum term (usually close to 1),  $\mu_{\text{prev}}$  and  $\sigma_{\text{prev}}^2$  are the accumulated means and variances from previous mini-batches, and  $\mu_B$  and  $\sigma_B^2$  are the current mini-batch mean and variance.

At test time, for any given input  $z^{(\text{test})}$ , the normalization is performed using the estimated  $\mu$  and  $\sigma^2$ :

$$\hat{z}^{(\text{test})} = \frac{z^{(\text{test})} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

And then the normalized test input is scaled and shifted using the learned  $\gamma$  and  $\beta$ :

$$z_{\text{norm}}^{(\text{test})} = \gamma \hat{z}^{(\text{test})} + \beta$$



## 8. Multi-class Classification:

To explain Softmax regression with math, let's break it down step by step:

1. **Linear Function:** In a neural network, for the final layer (let's call it layer L), we first compute the linear combination of the inputs from the previous layer. This is given by:

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

Here,  $W^{[L]}$  is the weight matrix,  $a^{[L-1]}$  is the activation from the previous layer, and  $b^{[L]}$  is the bias vector for layer L. For the case of Softmax regression,  $z^{[L]}$  will be a  $C \times 1$  dimensional vector, where  $C$  is the number of classes.

2. **Softmax Activation Function:** The Softmax function is applied to  $z^{[L]}$  to obtain the output probabilities for each class. The Softmax function for the  $i$ -th element in  $z^{[L]}$  is given by:

$$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{j=1}^C e^{z_j^{[L]}}}$$

Here,  $e^{z_i^{[L]}}$  is the exponential of the  $i$ -th element of the vector  $z^{[L]}$ , and the denominator is the sum of exponentials of all elements in  $z^{[L]}$ , ensuring that the output probabilities sum to 1.

3. **Output Vector:** The output  $a^{[L]}$  (also denoted as  $\hat{y}$ ) is a  $C \times 1$  dimensional vector where each element  $a_i^{[L]}$  represents the probability that the input belongs to class  $i$ .

4. **Example Calculation:** Suppose  $z^{[L]}$  is a 4-dimensional vector with values [5, 2, -1, 3]. Applying the Softmax function, we first compute the exponential of each element:

$$e^5 = 148.4, e^2 = 7.4, e^{-1} = 0.37, e^3 = 20.1$$

The sum of these exponentials is  $148.4 + 7.4 + 0.37 + 20.1 = 176.27$ .

Then, each element of  $a^{[L]}$  is computed by dividing each exponential by the sum:

$$a_1^{[L]} = \frac{148.4}{176.27}, a_2^{[L]} = \frac{7.4}{176.27}, a_3^{[L]} = \frac{0.37}{176.27}, a_4^{[L]} = \frac{20.1}{176.27}$$

This process results in a probability distribution over the 4 classes, with each element of  $a^{[L]}$  representing the probability that the input belongs to one of the classes, ensuring these probabilities sum to 1. Softmax regression thus generalizes logistic regression to multi-class classification problems by transforming the linear outputs into probabilities suitable for classifying inputs into one of multiple categories.

## 9. Training a softmax classifier:

## Forward Propagation

1. **Compute the raw scores (logits):** For the final layer  $L$ , compute the raw scores  $z^{[L]}$  using the previous layer's activations  $a^{[L-1]}$ , weights  $W^{[L]}$ , and biases  $b^{[L]}$ .  

$$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$$
2. **Apply the softmax function:** The softmax function is applied to the raw scores to get the output probabilities  $a^{[L]}$  for each class. For each element  $z_i$  in  $z^{[L]}$ , the softmax function is:  

$$a_i^{[L]} = \frac{e^{z_i}}{\sum_{j=1}^C e^{z_j}}$$
  
where  $C$  is the number of classes, and the sum in the denominator runs over all classes to normalize the probabilities.

## Loss Function

The most common loss function used with softmax output is the cross-entropy loss. For a single training example with true label  $y$  (often in one-hot encoded form where one element is 1 and the rest are 0) and predicted probabilities  $\hat{y} = a^{[L]}$ , the loss is:

$$L(y, \hat{y}) = -\sum_{i=1}^C y_i \log(\hat{y}_i)$$

For multi-class classification, this simplifies to  $-\log(\hat{y}_{\text{true class}})$  because  $y_i$  is 0 for all classes except the true class.

## Cost Function

The cost function  $J$  is the average loss over all  $m$  training examples:

$$J = -\frac{1}{m} \sum_{k=1}^m \sum_{i=1}^C y_{k,i} \log(\hat{y}_{k,i})$$

where  $y_{k,i}$  is the true label for the  $i$ -th class of the  $k$ -th example, and  $\hat{y}_{k,i}$  is the predicted probability for the  $i$ -th class of the  $k$ -th example.

## Backpropagation

To update the model's parameters, you need to compute the gradients of the cost function with respect to the parameters. For the softmax layer, the gradient of the cost  $J$  with respect to the raw scores  $z^{[L]}$  is:

$$\frac{\partial J}{\partial z^{[L]}} = \hat{y} - y$$

This equation shows the difference between the predicted probabilities and the actual labels, which is used to adjust the weights and biases during training.

## Gradient Descent Update

Finally, the parameters are updated using gradient descent. For weights  $W^{[L]}$  and biases  $b^{[L]}$ , the updates are:

$$W^{[L]} := W^{[L]} - \alpha \frac{\partial J}{\partial W^{[L]}}$$

$$b^{[L]} := b^{[L]} - \alpha \frac{\partial J}{\partial b^{[L]}}$$

where  $\alpha$  is the learning rate, and the gradients of  $J$  with respect to  $W^{[L]}$  and  $b^{[L]}$  are computed using the chain rule from the gradient  $\frac{\partial J}{\partial z^{[L]}}$ .

By iteratively applying these steps across all training examples and for a number of training iterations (epochs), the neural network learns to minimize the cost function, thereby improving its predictions.

## 10. Deep learning framework: