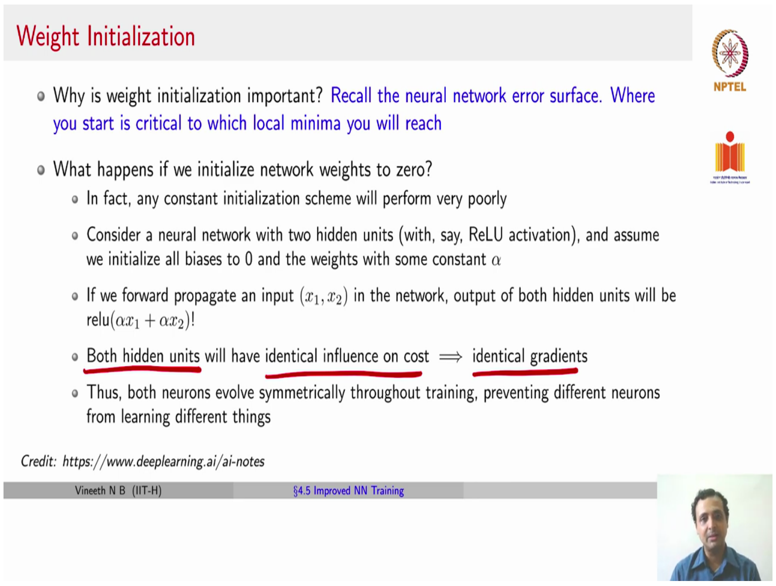
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 30
Improving Training of Neural Networks

(Refer Slide Time: 00:15)



Weight Initialization

- Why is weight initialization important? Recall the neural network error surface. Where you start is critical to which local minima you will reach
- What happens if we initialize network weights to zero?
  - In fact, any constant initialization scheme will perform very poorly
  - Consider a neural network with two hidden units (with, say, ReLU activation), and assume we initialize all biases to 0 and the weights with some constant α
  - If we forward propagate an input (x_1, x_2) in the network, output of both hidden units will be relu$(\alpha x_1 + \alpha x_2)$!
  - Both hidden units will have identical influence on cost $\Longrightarrow$ identical gradients
  - Thus, both neurons evolve symmetrically throughout training, preventing different neurons from learning different things

Credit: https://www.deeplearning.ai/ai-notes

Vineeth N B (IIT-H) §4.5 Improved NN Training

Another important component we are going to talk about here is weight initialization. You know by now that for a neural network to learn, you must first initialize the weights with certain values. And then you perform gradient descent to learn. Is this really important? Can I just initialize all of them with 0's and go?

It is important because if you recall the neural network error surface where you start is critical to which local minima you will reach. We saw that the error surface can be very complex which means it depends on which starting point you have. And you may accordingly converge to a local minima.

If you want to get to a better local minima, you may have to start at a better position initially to be able to get that solution. There have been recent studies which tried to analyse this from other perspectives. But weight initialization in general is considered a very important aspect of designing neural networks.

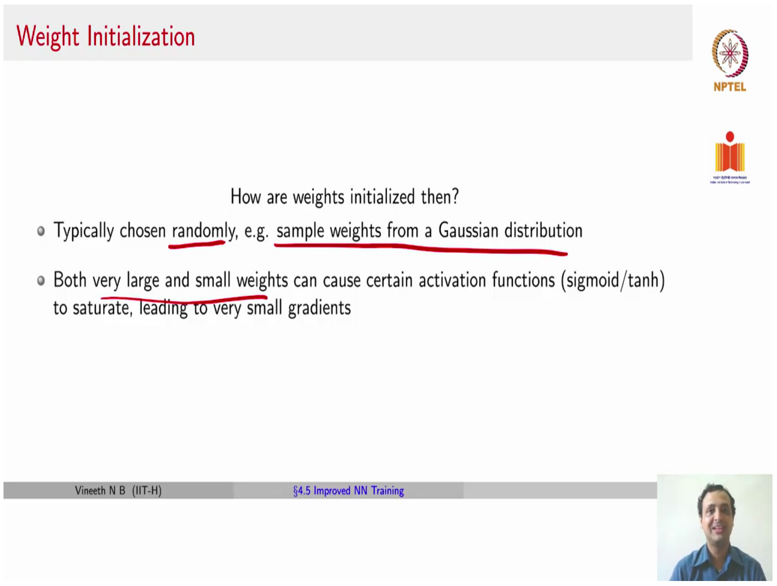
In fact, to a certain extent, this was a game changer when deep learning started becoming popular in the first decade of the 21st century. Let us first ask the trivial question: Why do not we just initialize the neural network weights to 0's and let it learn what it should learn? Is this a good answer? No. In fact, any constant weight initialization scheme will perform poorly. Why so? Let us take an example and understand this.

Let us consider a neural network with two hidden units. Let us say ReLU activation. And let us make all biases 0 just for us to understand this and let all the weights be initialized with some constant α. It could be zero; it could be non-zero; any constant initialization is okay here.

If we do this and forward propagate an input in the network, the output of both hidden units in the next layer will be $ReLU(\alpha x_1 + \alpha x_2)$ because the weights are the same. Now, why is this an issue? Which means both hidden units will have the same output. They will have an identical influence on the cost or the loss, which means they will have identical gradients.

If they have identical gradients, they will have identical updates and the weights will always remain the same. They start the same, the gradients will be the same and they will keep taking the same value always though training. This is not desirable. So, we want to do something more than simply initialize the weights with the same values.

(Refer Slide Time: 03:24)



Weight Initialization

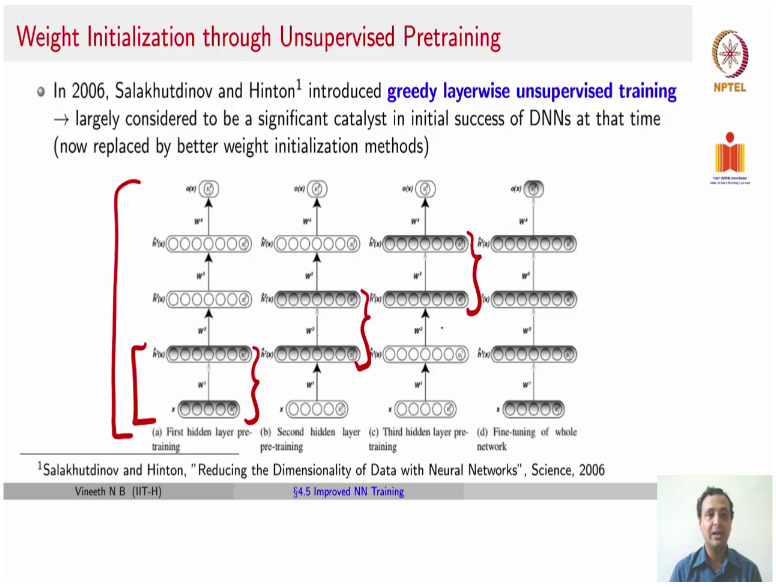
How are weights initialized then?

- Typically chosen randomly, e.g. sample weights from a Gaussian distribution
- Both very large and small weights can cause certain activation functions (sigmoid/tanh) to saturate, leading to very small gradients

Vineeth N B (IIT-H) §4.5 Improved NN Training

How are weights initialized then? They are generally chosen randomly. Or for example, you can sample weights from a Gaussian distribution. You have some variety in the weights. Both very large and small weights can cause activation functions such as sigmoid and tanh to saturate. So, that is something important to keep in mind when you initialize. Let us see more specifics on how we can initialize.

(Refer Slide Time: 03:59)



Weight Initialization through Unsupervised Pretraining

- In 2006, Salakhutdinov and Hinton[1] introduced **greedy layerwise unsupervised training** → largely considered to be a significant catalyst in initial success of DNNs at that time (now replaced by better weight initialization methods)

(a) First hidden layer pre-training
(b) Second hidden layer pre-training
(c) Third hidden layer pre-training
(d) Fine-tuning of whole network

[1]Salakhutdinov and Hinton, "Reducing the Dimensionality of Data with Neural Networks", Science, 2006

Vineeth N B (IIT-H) §4.5 Improved NN Training

In 2006, weight initialization was one of the major factors in turning around how deep neural networks were paid attention to. Salakhutdinov and Hinton introduced a method known as greedy layerwise unsupervised pre-training which is considered to be a significant catalyst in the training and success of deep neural networks.

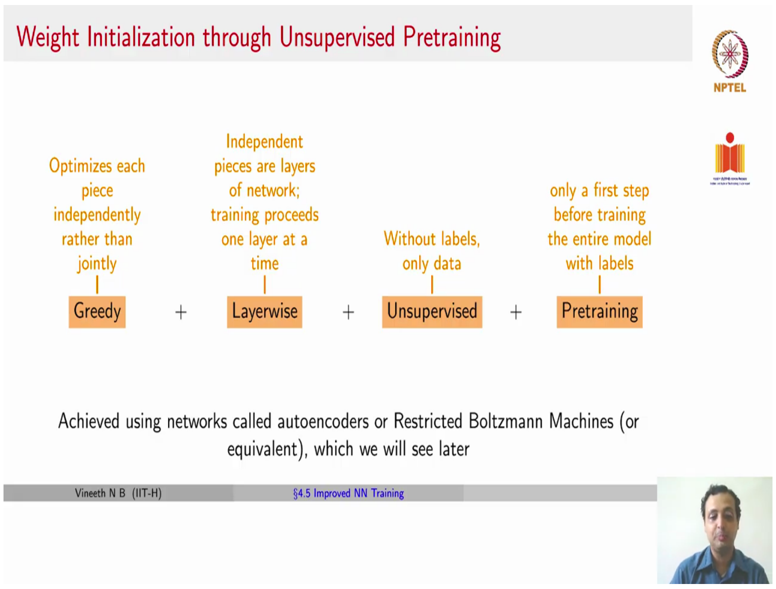
For a few years since 2006, the general understanding was this was extremely essential to make deep neural networks work in practice and led to widespread adoption of deep neural networks for various applications. However, today it is understood that you do not need this method. That there are simpler weight initialization methods that you can use and you get good results. But let us try to understand what this method tried to do.

As the name says, greedy layer wise, unsupervised pre-training. The suggestion was when you have a deep neural network, before you start training the neural network we are talking about the weight initialization step here, you take the first two layers and train them using some unsupervised technique. For example, you could use methods such as Boltzmann machines or auto encoders. We will see that later to be able to train these two layers in an unsupervised way.

Using that, you would get a set of weights. Now, you freeze those weights and then take the next two layers and train them in an unsupervised manner. Because the first layer's weights have frozen, you could provide inputs and multiply it by those frozen weights, and they will become inputs to the second layer.

So, you could take the second and third layer, train them in an unsupervised manner. And you would get a certain set of weights. Then you freeze those weights. Then you take the third set of third layer's weights, and train them in an unsupervised manner. And you keep repeating this process to get an initialization on your weights for the neural network.

(Refer Slide Time: 06:27)



Weight Initialization through Unsupervised Pretraining

Greedy	+	Layerwise	+	Unsupervised	+	Pretraining
Optimizes each piece independently rather than jointly		Independent pieces are layers of network; training proceeds one layer at a time		Without labels, only data		only a first step before training the entire model with labels

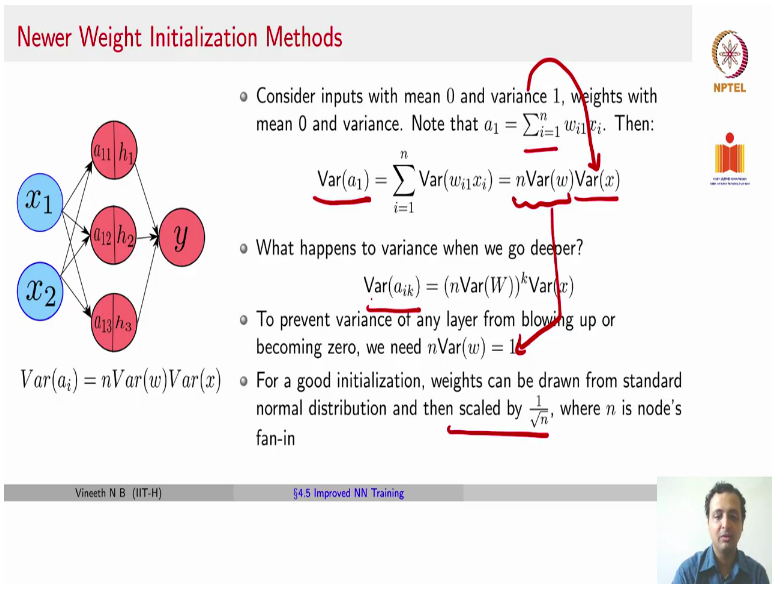
Achieved using networks called autoencoders or Restricted Boltzmann Machines (or equivalent), which we will see later

Vineeth N B (IIT-H) §4.5 Improved NN Training

Why is this called so? Greedy because it optimizes each piece independently rather than jointly. Layer wise because you are doing initialization in a layer wise manner. Unsupervised, as I just said, is because you use only the data, no labels to be able to train those networks. Because this is not a formal training process. It is only to initialize the neural network. And it is called pre-training because this is done before the formal training process starts. As I mentioned, it is for weight initialization.

So, a question that you may have now is how do you do that unsupervised training between every pair of layers? You may have to wait for that answer in this course. This is achieved by networks known as autoencoders or Restricted Boltzmann Machines. Or you could even use other methods to be able to achieve this. But we will see these examples; we will see autoencoders and RBMs, a little later in this course.

(Refer Slide Time: 07:35)



Newer Weight Initialization Methods

- Consider inputs with mean 0 and variance 1, weights with mean 0 and variance. Note that $a_1 = \sum_{i=1}^{n} w_{i1}x_i$. Then:

$$\text{Var}(a_1) = \sum_{i=1}^{n} \text{Var}(w_{i1}x_i) = n\text{Var}(w)\text{Var}(x)$$

- What happens to variance when we go deeper?

$$\text{Var}(a_{ik}) = (n\text{Var}(W))^k\text{Var}(x)$$

- To prevent variance of any layer from blowing up or becoming zero, we need $n\text{Var}(w) = 1$
- For a good initialization, weights can be drawn from standard normal distribution and then scaled by $\frac{1}{\sqrt{n}}$, where n is node's fan-in

$Var(a_i) = nVar(w)Var(x)$

Vineeth N B (IIT-H) §4.5 Improved NN Training

But as I mentioned, this got superseded by newer weight initialization methods over the last decade. Let us try to understand what has been the thought process behind coming up with newer initialization methods. If you took a neural network such as what you see here, let us consider that all your inputs have been normalized already. With mean 0 and variance 1. It is very common these days to normalize your data inputs before providing it as input to any machine learning algorithm. Let us also assume now that your weights have to have mean 0 and a certain variance. We do not know what that variance should be. That is what we want to find.

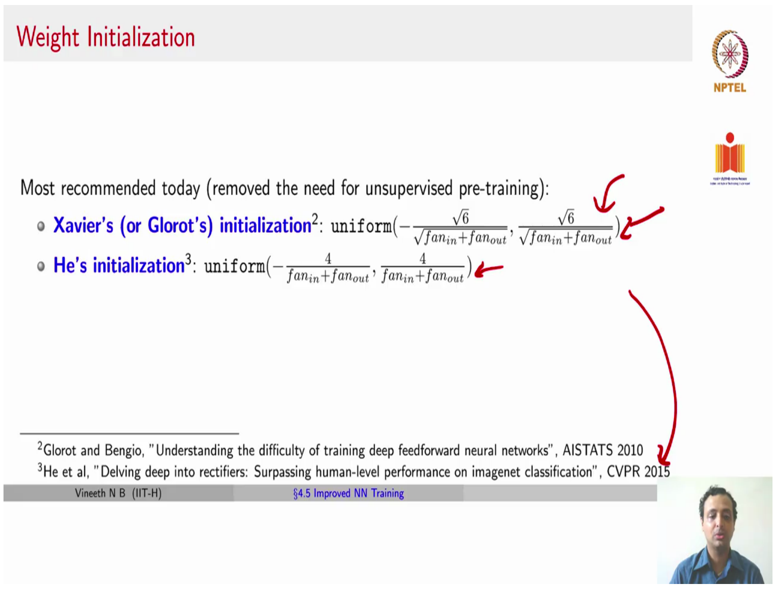
Let us now take one of these neurons. Let us call a_1. Let us take one of the neurons here. Let us call that a_1 for simplicity. That is going to be given by $\sum_{i=1}^{n} w_{i1}x_i$. Then the variance of a_1 would be $\sum_{i=1}^{n} Var(w_{i1}x_i)$.

Assuming now the w and x are not correlated with each other. You can write it as $nVar(w)Var(x)$. The joint covariance would go to 0. And assuming that we also want all neurons to have the same variance, the summation turns into n. Let us now try to understand what happens to the variance when we go deeper in the neural network. The variance would keep getting multiplied, and you would have the variance of a pre activation at the later layer to be $(nVar(w))^k Var(x)$. This could result in blowing up or becoming 0, depending on what the variance of w originally was.

Remember now that we ideally want, we already know the variance of x is given to be 1. We would want the variance of every succeeding layer also to be 1 because that would ensure that even the layers in between are normalized. So, we would ideally want the variance of a_1 also to be 1. The way to do it is that $nVar(w) = 1$. We know the variance of x is 1. We want variance of a_1 to also be 1 because then even the variances of activations in later layers will be normalized.

And then $nVar(w) = 1$. So, which means for a good initialization, you can draw weights from a normal distribution, a Gaussian distribution, and scale them by $1/\sqrt{n}$, where n is the node's fan-in. That is the same thing here. By fan-in we mean the number of weights coming into a particular neuron.

(Refer Slide Time: 11:04)



Weight Initialization

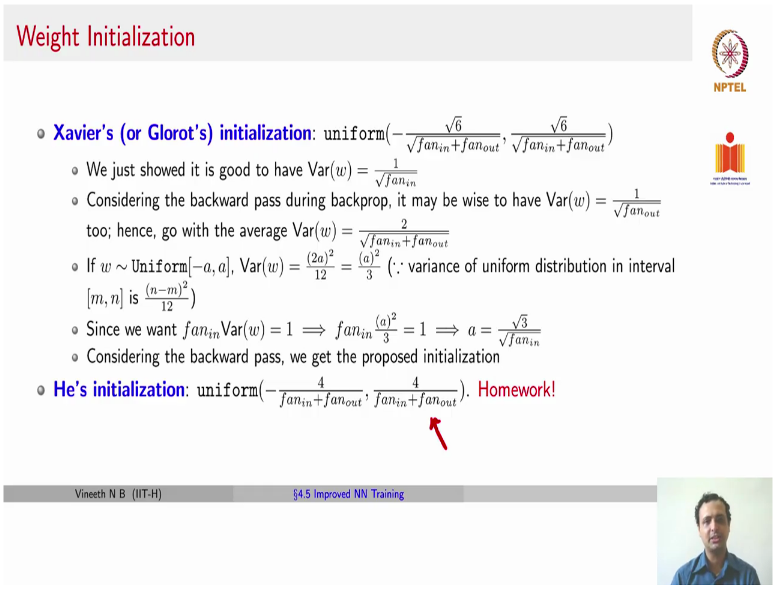
Most recommended today (removed the need for unsupervised pre-training):

- **Xavier's (or Glorot's) initialization**[2]: uniform$\left(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right)$
- **He's initialization**[3]: uniform$\left(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}}\right)$

[2]Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTATS 2010

[3]He et al, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", CVPR 2015

Vineeth N B (IIT-H) §4.5 Improved NN Training



Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$
  - We just showed it is good to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$
  - Considering the backward pass during backprop, it may be wise to have $\text{Var}(w) = \frac{1}{\sqrt{fan_{out}}}$ too; hence, go with the average $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$
  - If $w \sim \texttt{Uniform}[-a, a]$, $\text{Var}(w) = \frac{(2a)^2}{12} = \frac{(a)^2}{3}$ ($\because$ variance of uniform distribution in interval $[m, n]$ is $\frac{(n-m)^2}{12}$)
  - Since we want $fan_{in}\text{Var}(w) = 1 \implies fan_{in}\frac{(a)^2}{3} = 1 \implies a = \frac{\sqrt{3}}{\sqrt{fan_{in}}}$
  - Considering the backward pass, we get the proposed initialization
- **He's initialization**: $\texttt{uniform}(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}})$. Homework!

Vineeth N B (IIT-H) §4.5 Improved NN Training

Using the skew, there have been a few weight initialization methods that have been developed. The most popular ones today are Xavier's initialization or Glorot's initialization, which is given by this down here. You uniformly sample from a certain range given by these values or timing these initializations where you uniformly sample in this range. These came from these two papers which were published in 2010 and 2015. Let us try to understand one of them at least in detail as to how that range came through.

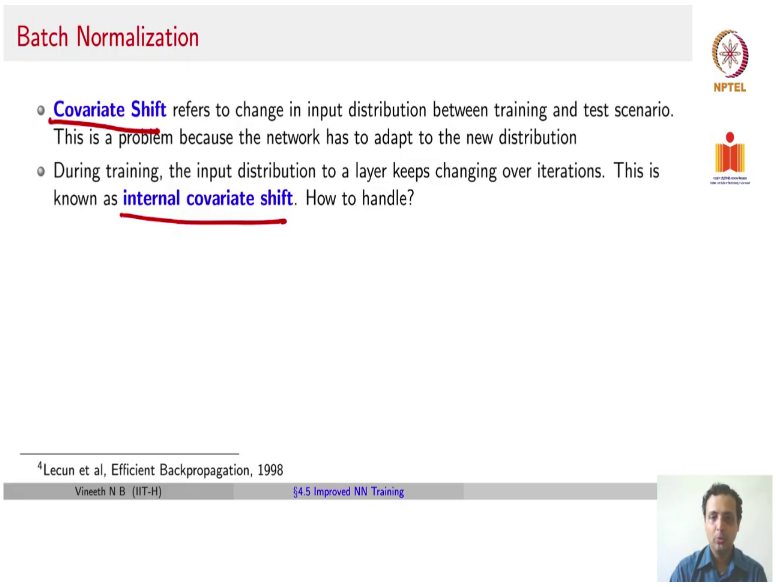
Let us not derive how Xavier's or Glorot's initialization was derived. We just now showed that it is good to have the variance of w to be $1/\sqrt{fan_{in}}$. But we also have to consider the backward pass during backprop which also could result in affecting the weights and the gradients. So even the backward pass could have an impact when you have to design the variance of the gradients because the gradients are distributed based on how the weights in a particular layer are, which flows through the previous layers, and so on and so forth.

So, it may be wise to also consider variance of w to be $a/\sqrt{fan_{out}}$. Fan-out is the number of weights going out of a neuron for the next layer. Let us hence go with the average variance of w to be $2/\sqrt{fan_{in} + fan_{out}}$. Now we know from statistics of uniform distribution that given a uniform distribution from the range, m, n; we know the variance is given by $(n - m)^2/12$. Which means if we now sample uniformly from minus $[- a, a]$, its variance is going to be given by $(2a)^2/12$ which is $a^2/3$.

Now, let us put these together. We said that $fan_{in} * Var(w)$must be 1. Fan-in is the same as n of the previous slide. We know that the variance of w coming from a uniform distribution from an interval is $a^2/3$. Putting these two together, $fan_{in} * a^2/3 = 1$or $a = \sqrt{3}/\sqrt{fan_{in}}$. And now considering the backward pass and the forward pass, you get Xavier's initialization to be uniform($-\sqrt{6}/(\sqrt{fan_{in} + fan_{out}}), \sqrt{6}/(\sqrt{fan_{in} + fan_{out}})$).

Kaiming initialization is built on top of this. But that is going to be homework for you to work out as to how you get this as another possible initialization. You can by looking at it say that the ideas have to be similar, but there are some minor changes. You can also look at the paper that proposed this, which was in the footnote on the previous slide, to be able to do this homework.

(Refer Slide Time: 14:18)

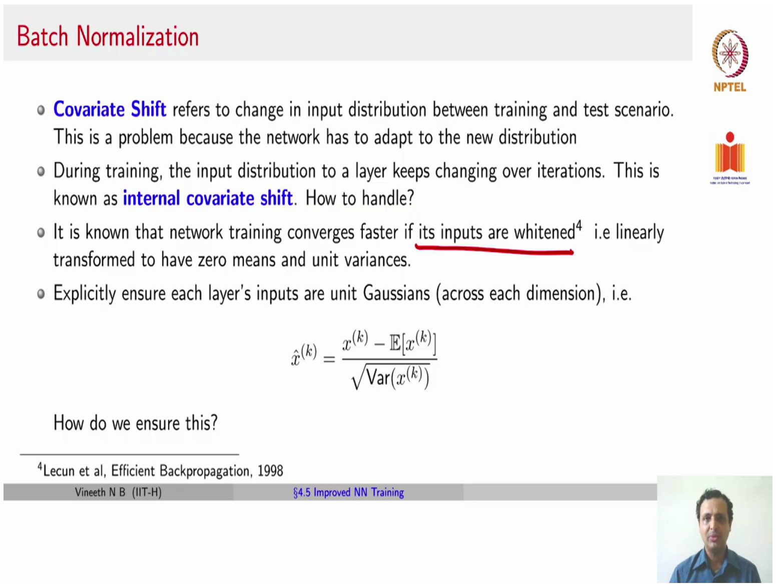


Batch Normalization

- Covariate Shift refers to change in input distribution between training and test scenario. This is a problem because the network has to adapt to the new distribution
- During training, the input distribution to a layer keeps changing over iterations. This is known as internal covariate shift. How to handle?

[4]Lecun et al, Efficient Backpropagation, 1998

Vineeth N B (IIT-H) §4.5 Improved NN Training



Batch Normalization

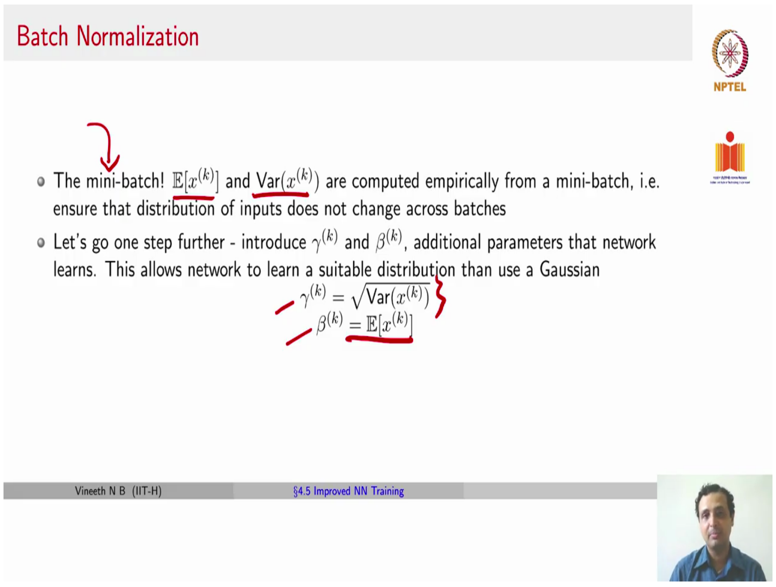
- Covariate Shift refers to change in input distribution between training and test scenario. This is a problem because the network has to adapt to the new distribution
- During training, the input distribution to a layer keeps changing over iterations. This is known as internal covariate shift. How to handle?
- It is known that network training converges faster if its inputs are whitened[4] i.e linearly transformed to have zero means and unit variances.
- Explicitly ensure each layer's inputs are unit Gaussians (across each dimension), i.e.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}(x^{(k)})}}$$

How do we ensure this?

[4]Lecun et al, Efficient Backpropagation, 1998

Vineeth N B (IIT-H) §4.5 Improved NN Training



Batch Normalization

- The mini-batch! $\mathbb{E}[x^{(k)}]$ and $\text{Var}(x^{(k)})$ are computed empirically from a mini-batch, i.e. ensure that distribution of inputs does not change across batches
- Let's go one step further - introduce $\gamma^{(k)}$ and $\beta^{(k)}$, additional parameters that network learns. This allows network to learn a suitable distribution than use a Gaussian

$$\gamma^{(k)} = \sqrt{\text{Var}(x^{(k)})}$$
$$\beta^{(k)} = \underline{\mathbb{E}[x^{(k)}]}$$

Vineeth N B (IIT-H) §4.5 Improved NN Training

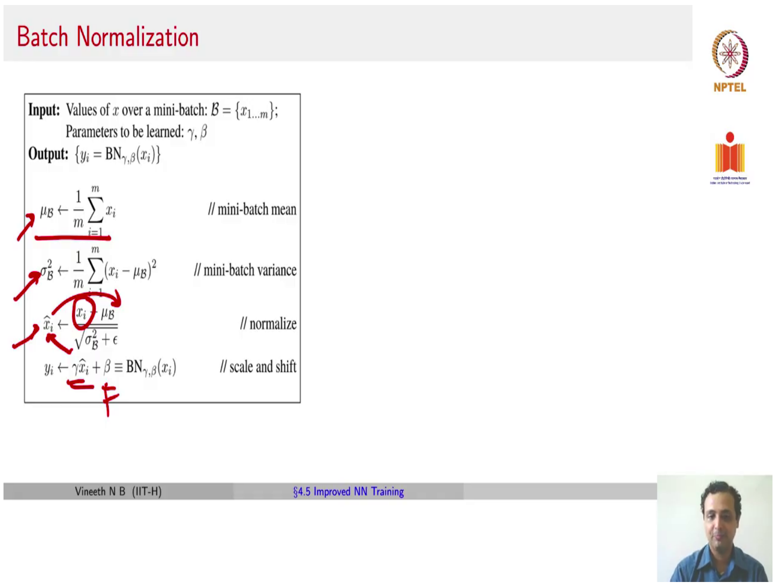
Let us move to the last component of this lecture that we are going to talk about which is an important development that came about in 2015 known as batch normalization. Covariate shift is a problem that in machine learning refers to a change in input distribution between training and test scenarios. This generally causes problems because the model needs to adapt to a new distribution. This could also happen even during the training process itself between the distribution at one stage and the distribution at another stage.

In a neural network, this issue can show up as internal covariate shift where the input distribution can change across the layers. So, the first layer we could normalize the data and it would have the inputs would follow a certain distribution. Depending on the weights in the first layer, the second layer would receive a different distribution. The third layer would receive a different distribution based on those weights, and so on and so forth. And the neural network has to learn to handle these different distributions. Can we do something to address this?

We know that a network trains well when its inputs are widened or normalized. They are linearly transformed to have 0 means and unit variances. Can we now do something to also have a similar effect in each layer? Can we make each layer also a unit Gaussian? That is not in our hands entirely because it depends on the weights. But how do we achieve such an effect? The answer to this is, we could do this by considering a mini-batch.

Since we train neural networks using mini-batch SGD, we could try to see if we could compute the mean and variance of any layers or outputs in a mini-batch. And then come to subtract the mean and divide by the variance to normalize them in some way. To do this, this method known as batch normalization, which was proposed in 2015 introduces 2 additional parameters: gamma and beta. The superscript k denotes the specific layer in which you introduce these parameters because you have to do layerwise. $\gamma = \sqrt{Var(x^{(k)})}$ and $\beta = E[x^{(k)}]$.

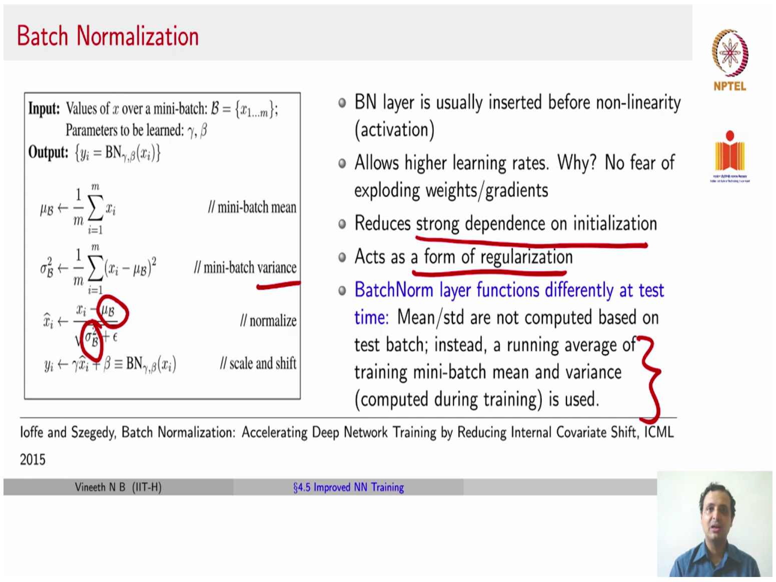
(Refer Slide Time: 16:59)


Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

```
$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                  // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$            // scale and shift
```

Vineeth N B (IIT-H) §4.5 Improved NN Training


Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

```
$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                  // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$            // scale and shift
```

- BN layer is usually inserted before non-linearity (activation)
- Allows higher learning rates. Why? No fear of exploding weights/gradients
- Reduces strong dependence on initialization
- Acts as a form of regularization
- BatchNorm layer functions differently at test time: Mean/std are not computed based on test batch; instead, a running average of training mini-batch mean and variance (computed during training) is used.

Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Vineeth N B (IIT-H) §4.5 Improved NN Training

Let us look at how this batch normalization works. Batch normalization is often introduced as a layer itself that succeeds one of the layers, one of the existing layers of a neural network. So, let us consider this. We assume that after a certain layer you have certain output values that you get. Let the mean of all of them across the mini-batch that you are propagating in a certain SGD iteration, let that be μ_B.

Similarly, you can compute the mini-batch variance across those values that you are considering, across that layer that you are considering to be σ_B^2. Now, if we normalize your $\hat{x}_i = (x_i - \mu_B)/(\sigma_B^2 + \epsilon)$.

What we do is to say that the output of this batch normalization layer, $y_i = \gamma \hat{x}_i + \beta$, where γ and β are learned.

How does this help? Once you learn γ and β, in a sense the output of that layer is renormalized. The mean, the new so-called mean would be β and γ would be the standard deviation of this new distribution. And by learning γ and β, you are asking the neural network, normalize each layer the way you want to normalize it. You do not need to do standard normalization. That means 0 and unit variance. Choose the mean, choose the variance that helps you perform the best and let the neural network learn γ and β.

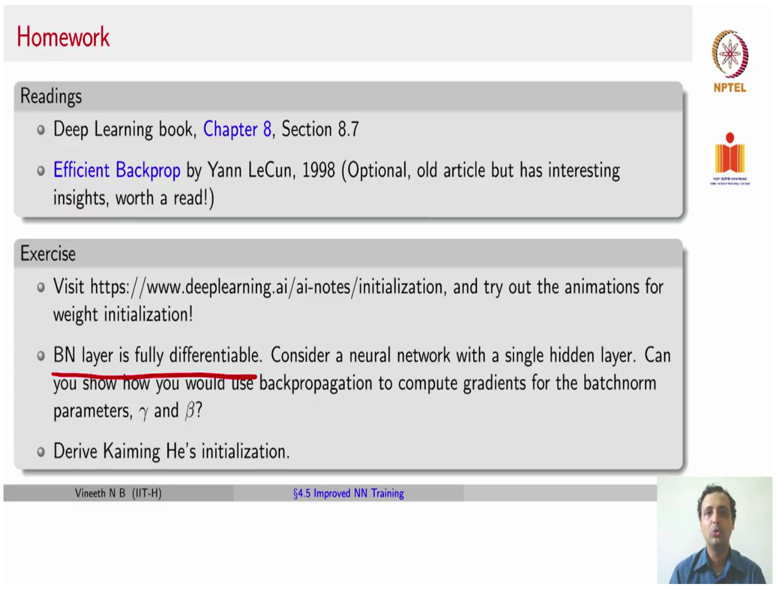
The Batch Norm layer is generally inserted before you apply your non-linearity. So, you have a layer which receives inputs from the previous layer, you do batch normalization, and then apply your non-linearity. This is typically the way it is implemented. And this has given good empirical results. Batch normalization allows higher learning rates. Why?

One of the reasons to keep your learning rates low was to avoid your weights to explode in a neural network or on the other side if it becomes too low. But we are talking about higher learning rates here. By ensuring that your activations are controlled because of the batch normalization, the learning rates can now be increased because this learning the γ and β will take care of what the neural network needs to not let it explode.

It reduces a strong dependence on initialization. Because now, irrespective of the initialization, the weights can be controlled using the γ and β in every successive iteration of SGD. It also acts as a form of regularization. Because whatever weights come or whatever activations come out of a layer, you are going to multiply by a γ and β, which has a sense of adding noise to those activations which becomes the regularizer.

An important difference, an important concern of batch normalization is that at test time, your data may not be forward propagated in batches. You may want to predict only on a single point. Then you may not have a mean and variance for a mini-batch at test time. How do you handle it? You choose μ_B and σ_B^2 based on your training data. Maybe the last few training batches or you take a running average of μ_B and σ_B^2 over a set of training iterations, and you use those values at testing.

(Refer Slide Time: 21:15)



Homework

Readings

- Deep Learning book, Chapter 8, Section 8.7
- Efficient Backprop by Yann LeCun, 1998 (Optional, old article but has interesting insights, worth a read!)

Exercise

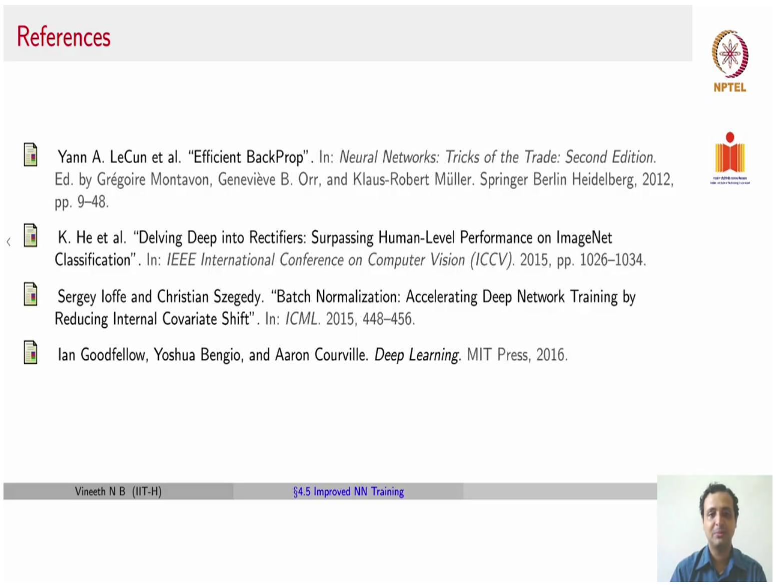
- Visit https://www.deeplearning.ai/ai-notes/initialization, and try out the animations for weight initialization!
- BN layer is fully differentiable. Consider a neural network with a single hidden layer. Can you show how you would use backpropagation to compute gradients for the batchnorm parameters, γ and β?
- Derive Kaiming He's initialization.

Vineeth N B (IIT-H) §4.5 Improved NN Training

That concludes our discussion of batch normalization. Your recommended readings for this lecture are chapter 8, section 8.7 of the deep learning book. And a very nice article called Efficient Backprop by Yann LeCun. It is a very old article. Some of it is not relevant to deep neural networks but some of it has very nice insights on efficient training of neural networks. I would recommend you to read it.

Your exercises are visit this link here which has some nice animations of how a neural network behaves with different initializations. It is a simple "click and try" exercise for you. The second exercise is to prove batch normalization is fully differentiable. Can you try finding out how you would compute gradients for the Batch Norm layer? It should be by looking at it you can make all that it is simply a multiplicative factor and an additive factor which should be differentiable. But how would you get $\partial L/\partial \gamma$ and $\partial L/\partial \beta$. Please work it out. And as we already left behind, derive Kaiming He's initialization.

(Refer Slide Time: 22:32)



References

Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition.* Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Springer Berlin Heidelberg, 2012, pp. 9–48.

K. He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *IEEE International Conference on Computer Vision (ICCV).* 2015, pp. 1026–1034.

Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *ICML.* 2015, 448–456.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

Vineeth N B (IIT-H) §4.5 Improved NN Training

Here are some references.