Deep Learning for Computer Vision

Improvements in Training of Neural Networks

Vineeth N Balasubramanian

Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

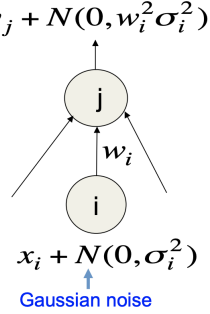
- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer
- In a simple net with a linear output unit directly connected to inputs, the amplified noise gets added to output

Review

Exercise

- Show that adding Gaussian noise (with zero mean) to input is equivalent to L_2 weight decay when loss function is MSE

- When we add Gaussian noise to inputs, variance of noise is amplified by squared weight before going to next layer
- In a simple net with a linear output unit directly connected to inputs, the amplified noise gets added to output
- This makes an additive contribution to the squared error, i.e. minimizing squared error tends to minimize squared weights when inputs are noisy!



Credit: Tijmen Tieleman, Univ of Toronto

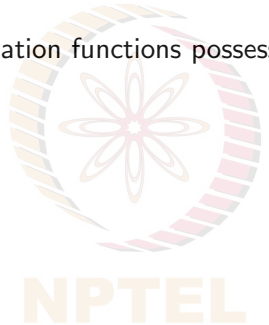
Review

Mathematically, consider one input with added noise. We would have:
$y_{\text{noisy}} = \sum_i w_i x_i + \sum_i w_i \epsilon_i$, where ϵ_i is sampled from $N(0, \sigma_i^2)$. Given expected output t, we have:

$$\begin{aligned}
\mathbb{E}\left[\left(y_{\text{noisy}} - t\right)^2\right] &= \mathbb{E}\left[\left(y + \sum_i w_i\epsilon_i - t\right)^2\right] = \mathbb{E}\left[\left((y-t) + \sum_i w_i\epsilon_i\right)^2\right] \\
&= (y-t)^2 + \mathbb{E}\left[2(y-t)\sum_i w_i\epsilon_i\right] + \mathbb{E}\left[\left(\sum_i w_i\epsilon_i\right)^2\right] \\
&= (y-t)^2 + \mathbb{E}\left[\left(\sum_i w_i\epsilon_i\right)^2\right] \\
&\left(\because \epsilon_i \text{ is independent of } \epsilon_j, \text{ and } \epsilon_i \text{ is independent of } (y-t)\right) \\
&= (y-t)^2 + \sum_i w_i^2\sigma_i^2 \implies \text{Penalty on L2-norm of weights!}
\end{aligned}$$

Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?



Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?
- Must be continuous, differentiable (need to perform backpropagation), non-decreasing and easy to compute.

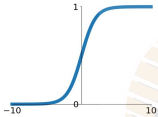
Activation Functions

- Non-linear function applied to the output of a neuron.
- What characteristics must activation functions possess?
- Must be continuous, differentiable (need to perform backpropagation), non-decreasing and easy to compute.
- Common activation functions available- Sigmoid, Tanh, ReLU, etc.
  - Which one to choose?
- What effect does a chosen activation function have on training?

Activation Functions

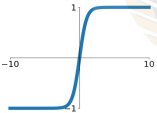
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



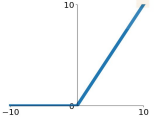
tanh

$$\tanh(x)$$



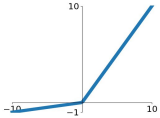
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

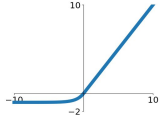


Maxout

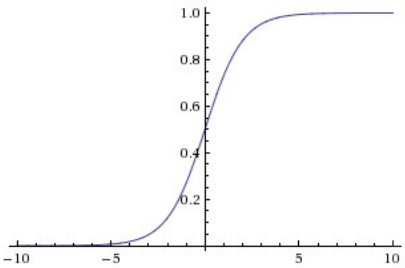
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



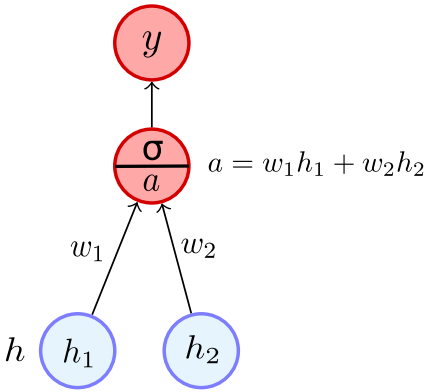
Activation Functions: Sigmoid



Sigmoid

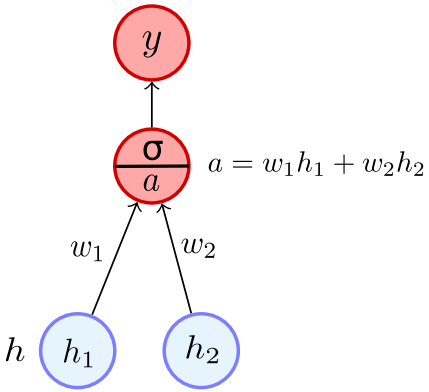
- $\sigma(x) = \frac{1}{1+e^{-x}}$
- Compresses all inputs to range $[0, 1]$
- Gradient of sigmoid function is: $\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$
- Sigmoid activations are rarely used in vanilla NNs today, and more often in specific architectures. Why?

Activation Functions: Sigmoid



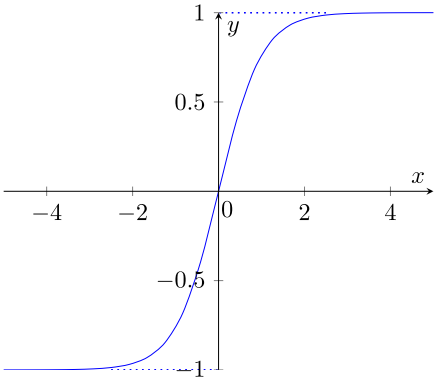
- Gradient of sigmoid neuron at saturation is very small $\implies$ training is slow
- Sigmoid is not zero-centered
- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_1$
  $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_2$
  If h_1 and h_2 are outputs of sigmoid neurons, they are positive. All gradients at a layer are either positive or negative.
- Computationally expensive, need to compute exponential function

Activation Functions: Sigmoid



- Gradient of sigmoid neuron at saturation is very small $\implies$ training is slow
- Sigmoid is not zero-centered
- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_1$
  $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \sigma}\frac{\partial \sigma}{\partial a}h_2$
  If h_1 and h_2 are outputs of sigmoid neurons, they are positive. All gradients at a layer are either positive or negative.
- Computationally expensive, need to compute exponential function

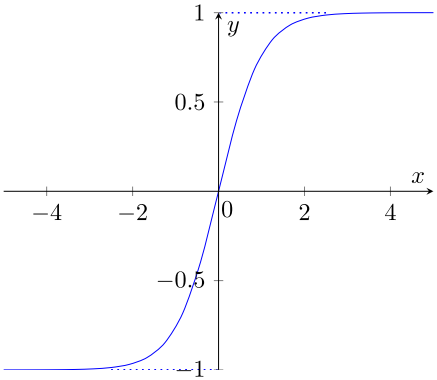
Activation Functions: Tanh



f(x) = tanh(x)

- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Compresses all inputs to range $[-1, 1]$
- Gradient of tanh function is: $\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x))$
- What advantage does it have over sigmoid?

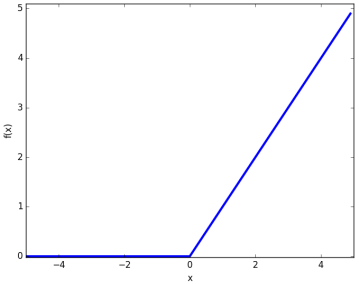
Activation Functions: Tanh



f(x) = tanh(x)

- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Compresses all inputs to range $[-1, 1]$
- Gradient of tanh function is: $\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x))$
- What advantage does it have over sigmoid? It is zero-centered
- However, gradient of tanh at saturation also vanishes
- Also computationally expensive due to the exponential

Activation Functions: Rectified Linear Unit (ReLU)



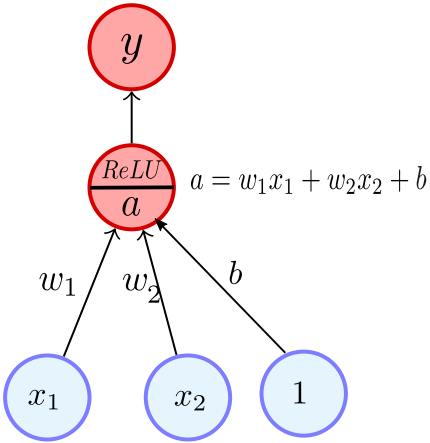
$$f(x) = max(0, x)$$

- Default activation used today
- ReLU computes $f(x) = \max(0, x)$
- Gradient of ReLU is:

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

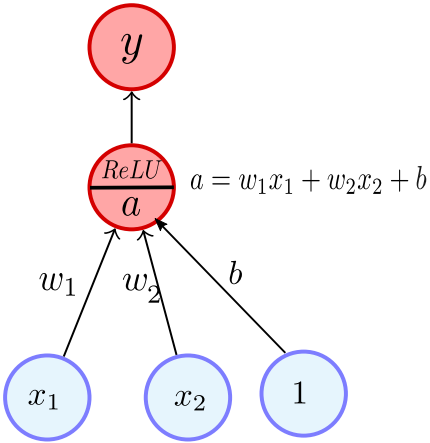
- Non-saturating for positive inputs
- Found to accelerate convergence of SGD compared to sigmoid/tanh functions (by a factor of 6 in AlexNet)
- Computationally inexpensive

The Dying ReLU Problem



- If input to ReLU neuron is negative, output is zero ("**dead neuron**"). How do the gradients flow?
  $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial ReLU} \frac{\partial ReLU}{\partial a} x_1$
- if $a < 0$, gradient term $\frac{\partial ReLU}{\partial a}$ is zero, which makes whole gradient zero. Thus, weights w_1 and w_2 do not get updated

The Dying ReLU Problem



- If input to ReLU neuron is negative, output is zero ("**dead neuron**"). How do the gradients flow?
  $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial ReLU} \frac{\partial ReLU}{\partial a} x_1$
- if $a < 0$, gradient term $\frac{\partial ReLU}{\partial a}$ is zero, which makes whole gradient zero. Thus, weights w_1 and w_2 do not get updated
- Initializing bias to small positive value can help
- Using variants of ReLU (Leaky ReLU, Exponential Linear Unit) can help

Activation Functions: Which one to choose?

- Use ReLU non-linearity, be careful with learning rates and monitor the fraction of 'dead' units in a network.
- Try Leaky ReLU, ELU, Maxout
- Try tanh, but expect worse performance
- Sigmoid not used much, unless a gating is required
- In general, a good idea for an activation function to be in its linear region for most part of training

Credit: Fei-Fei Li, CS231N, Stanford Univ

Weight Initialization

- Why is weight initialization important?

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?

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

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$(\alpha x_1 + \alpha x_2)$!

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 $\implies$ identical gradients
  - Thus, both neurons evolve symmetrically throughout training, preventing different neurons from learning different things

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

Weight Initialization

How are weights initialized then?

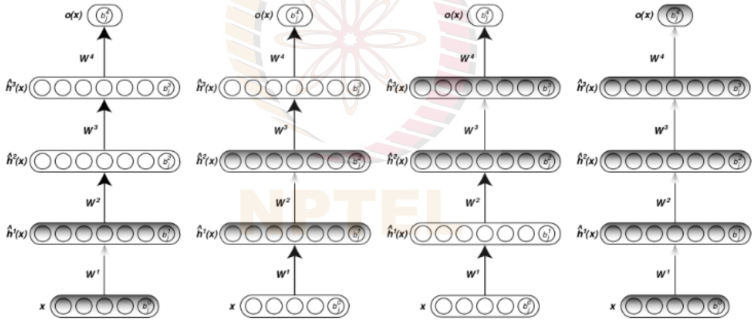
Weight Initialization

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

Weight Initialization through Unsupervised Pretraining

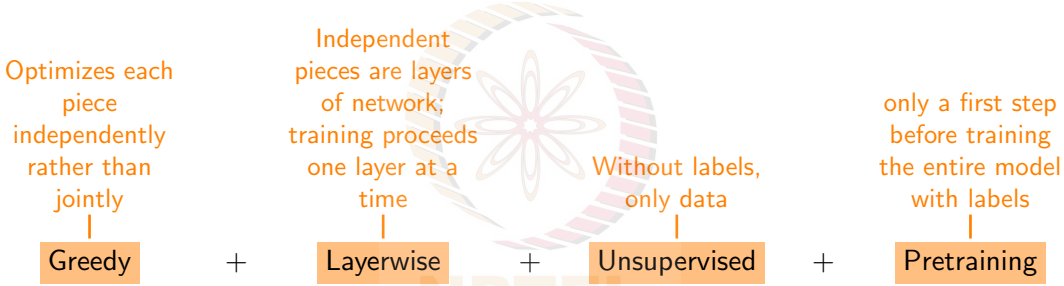
- In 2006, Salakhutdinov and Hinton[1] introduced **greedy layerwise unsupervised training** → largely considered to be a significant catalyst in initial success of DNNs at that time (now replaced by better weight initialization methods)



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

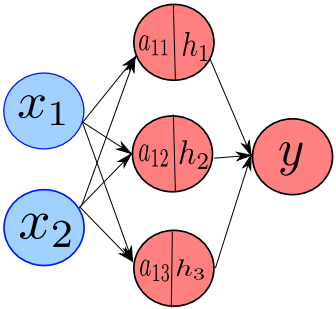
[1] Salakhutdinov and Hinton, "Reducing the Dimensionality of Data with Neural Networks", Science, 2006

Weight Initialization through Unsupervised Pretraining



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

Newer Weight Initialization Methods

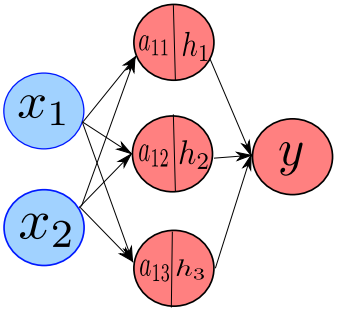


$$Var(a_i) = nVar(w)Var(x)$$

- 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:

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

Newer Weight Initialization Methods



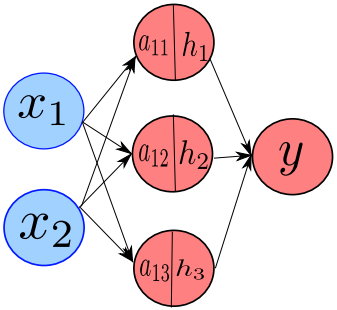
$$Var(a_i) = nVar(w)Var(x)$$

- 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?

Newer Weight Initialization Methods



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

- 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

Weight Initialization

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

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

[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

Weight Initialization

- **Xavier's (or Glorot's) initialization**: $\texttt{uniform}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$

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 $\text{Var}(w) = \frac{1}{\sqrt{fan_{in}}}$

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 $\text{Var}(w) = \frac{2}{\sqrt{fan_{in}+fan_{out}}}$

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 $\frac{(n-m)^2}{12}$)

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 $fan_{in}\text{Var}(w) = 1 \implies fan_{in}\frac{(a)^2}{3} = 1 \implies a = \frac{\sqrt{3}}{\sqrt{fan_{in}}}$

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

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!

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

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

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

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

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

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

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- BN layer is usually inserted before non-linearity (activation)
- Allows higher learning rates. Why?

Batch Normalization

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

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- 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

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.

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.