Using Skip Connections to Mitigate the Problem of Vanishing Gradients in Deep Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Thursday 9$^{\text{th}}$ April, 2020   23:23

Preamble

Solving difficult image recognition and detection problems requires deep networks.

But training deep networks can be difficult because of the vanishing gradients problem. Vanishing gradients means that the gradients of the loss become more and more muted in the beginning layers of a network as the network become increasingly deeper.

The main reason for this is the multiplicative effect that goes into the calculation of the gradients — the gradient calculated for each layer are a product of the partial derivatives in all the higher indexed layers in the network.

Modern deep networks use multiple strategies to cope with the problem of vanishing gradients. These are listed on the next slide.

Preamble (contd.)

Here are the strategies commonly used used for addressing the problem of vanishing gradients:

- normalized initializations for the learnable parameters
- batch normalization
- using skip connections

Of the three mitigation strategies listed above, the last — using skip connections — is the most counter intuitive and, therefore, also the most fun to talk about. So I'll take it up first in this lecture.

This lecture will start with a demonstration of the improvement in classification accuracy when using skip connections. For this demo, I'll use the BMEnet inner class of my DLStudio-1.0.6 module.

Subsequently, I'll explain more precisely what is meant by the vanishing gradient problem in deep learning and why using skip connections helps.

Outline

Outline

SkipConnection Class in My DLStudio Module

- I think the best way to start this lecture is with an in-class demonstration of the improvements in the performance of a CNN by using skip connections. I'll use my DLStudio module for the demo.

- You can install this module from:

    https://pypi.org/project/DLStudio/1.0.6

    Make sure that you have **Version 1.0.6 or higher** of the module.

- The in-class demo is based on the inner class SkipConnections of the module. This is just a convenience wrapper class for the actual network classes used in the demo: BMEnet and SkipBlock.

- **I have used "BME" in the name BMEnet in honor of the School of Biomedical Engineering, Purdue University.** Without their sponsorship, you would not be taking BME695DL/ECE695DL this semester.

The Roles Played by the BMEnet and SkipBlock Classes

- Just as `torch.nn.Conv2d`, `torch.nn.Linear`, etc., are the building blocks of a CNN in PyTorch, `SkipBlock` will serve as the primary building block for creating a deep network with skip connections.

- What I mean by that is that we will build a network whose layers are built from instances of `SkipBlock`.

- As to the reason for why the building block is named `SkipBlock`, it has two pathways for the input to get to the output: one goes through a couple of convolutional layers and other just directly. Obviously, the input going directly to the output means that it is skipping the convolutional layers in the other path.

- The overall network that is built with `SkipBlock` will be an instance of `BMEnet`.

Two Assumptions Implicit in the Definition of SkipBlock

- In the code shown on the next slide, an instance of SkipBlock consists of two convolutional layers, as you can see in lines (B) and (E). Each convolutional output is subject to batch normalization and ReLU activation. Additionally, this class is based on the following two implicit assumptions:

  1. In the overall network, when a convolutional kernel creates a lower-resolution output, the change will be by a factor of 2 exactly.
     [For example, a convolutional kernel may convert a 32×32 image into a 16×16 output, or a 16×16 input into an 8×8 output, and so on.]

  2. When the input and the output channels are unequal for a convolutional layer, the number of output channels is exactly twice the number of input channels.

- These two assumptions make it possible to define a small building block class that can then be used to create networks of arbitrary depth (without needing any additional glue code).

`SkipBlock`'s **Definition**

Can you see the skipping action in the definition of `forward()`? We combine the input saved in line (B) with the output at the end.

```
class SkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super(DLStudio.SkipConnections.SkipBlock, self).__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        norm_layer1 = nn.BatchNorm2d
        norm_layer2 = nn.BatchNorm2d
        self.bn1 = norm_layer1(out_ch)
        self.bn2 = norm_layer2(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

    def forward(self, x):                                               ## (A)
        identity = x                                                    ## (B)
        out = self.convo1(x)                                            ## (C)
        out = self.bn1(out)                                             ## (D)
        out = torch.nn.functional.relu(out)                            ## (E)
        if self.in_ch == self.out_ch:                                  ## (F)
            out = self.convo2(out)                                      ## (G)
            out = self.bn2(out)                                         ## (H)
            out = torch.nn.functional.relu(out)                        ## (I)
```

Continued on the next slide ....

`SkipBlock`'s **Definition (contd.)**

(...... continued from the previous slide)

```
if self.downsample:                                             ## (J)
    out = self.downsampler(out)                                 ## (K)
    identity = self.downsampler(identity)                       ## (L)
if self.skip_connections:                                       ## (M)
    if self.in_ch == self.out_ch:                               ## (N)
        out += identity                                         ## (O)
    else:                                                       ## (P)
        out[:,:self.in_ch,:,:] += identity                      ## (Q)
        out[:,self.in_ch:,:,:] += identity                      ## (R)
return out                                                      ## (S)
```

The BMEnet Class — Building a Network with SkipBlock

Now I am ready to define our main neural network class BMEnet:

```python
class BMEnet(nn.Module):

    def __init__(self, skip_connections=True, depth=32):
        super(DLStudio.SkipConnections.BMEnet, self).__init__()
        if depth not in [8, 16, 32, 64]:
            sys.exit("BMEnet has been tested for depth for only 16, 32, and 64")
        self.depth = depth // 8
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.append(DLStudio.SkipConnections.SkipBlock(64, 64, skip_connections=skip_connections
        self.skip64ds = DLStudio.SkipConnections.SkipBlock(64, 64, downsample=True, skip_connections=skip_conne
        self.skip64to128 = DLStudio.SkipConnections.SkipBlock(64, 128, skip_connections=skip_connections )
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(DLStudio.SkipConnections.SkipBlock(128, 128, skip_connections=skip_connecti
        self.skip128ds = DLStudio.SkipConnections.SkipBlock(128,128, downsample=True, skip_connections=skip_con
        self.fc1 = nn.Linear(2048, 1000)
        self.fc2 = nn.Linear(1000, 10)
```

The BMEnet Class (contd.)

(...... continued from the previous slide)

```
def forward(self, x):                                                       ## (A)
    x = self.pool(torch.nn.functional.relu(self.conv(x)))                   ## (B)
    for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):             ## (C)
        x = skip64(x)                                                       ## (D)
    x = self.skip64ds(x)                                                    ## (E)
    for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):             ## (F)
        x = skip64(x)                                                       ## (G)
    x = self.skip64ds(x)                                                    ## (H)
    x = self.skip64to128(x)                                                 ## (I)
    for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):           ## (J)
        x = skip128(x)                                                      ## (K)
    for i,skip128 in enumerate(self.skip128_arr[self.depth//4:]):           ## (L)
        x = skip128(x)                                                      ## (M)
    x = x.view(-1, 2048 )                                                   ## (N)
    x = torch.nn.functional.relu(self.fc1(x))                               ## (O)
    x = self.fc2(x)                                                         ## (P)
    return x                                                                ## (Q)
```

Explaining the BMEnet Class

- Here are the naming conventions I have used for the different types of SkipBlock instances used in BMEnet:

  - In the code in __init__(), when the name of a SkipBlock instance ends in "ds", as in the names self.skip64ds and self.skip128ds, that means that an important duty of that SkipBlock is to downsample the image array by a factor of 2.

  - The name self.skip64to128 is for an instance of SkipBlock that has 64 channels at its input and 128 channels at its output.

  - The names self.skip64 and self.skip128 are routine SkipBlock instances that neither downsample the images nor change the number of channels from input to output.

- The next slide talks about the forward() and its four for loops.

Explaining the BMEnet Class (contd.)

- As you can tell from the code in the forward() of BMEnet, I have divided the network definition into four sections, each section created with a for loop.

- The number of SkipBlock layers created in each section depends on the value given to self.depth.

- The four for loops are separated by either a downsmapling SkipBlock, which would either be self.skip64ds or self.skip128ds, or a channel changing one like self.skip64to128.

- The different SkipBlock versions are created by the two constructor options for the BMEnet class, skip_connections and depth that you see in the header of the __init__() for BMEnet.

Outline

The In-Class Demo

- The script `playing_with_skip_connections.py` in the Examples directory of the `DLStudio` distribution shows how you can ask the module constructor to create a `BMEnet` network with your choice of values for the options `skip_connections` and `depth`.

- The next slide shows the details of the network constructed when the constructor option `depth` is set to 32.

- Why do think the number of learnable parameters goes suddenly up from 1,792 in the layer `Conv2d-1` to 36,926 in the layer `Conv2d-3`?

- You will also notice that all the entries are zero for the `SkipBlock` layers. How do you explain that?

- At the end of the network detail, you will see that the total number of learnable parameters exceeds 5 million.

The Network Generated for `depth=32`

```
a summary of input/output for the model:
----------------------------------------------------------------
        Layer (type)            Output Shape          Param #
================================================================
           Conv2d-1          [-1, 64, 32, 32]           1,792
        MaxPool2d-2          [-1, 64, 16, 16]               0
           Conv2d-3          [-1, 64, 16, 16]          36,928
      BatchNorm2d-4          [-1, 64, 16, 16]             128
           Conv2d-5          [-1, 64, 16, 16]          36,928
      BatchNorm2d-6          [-1, 64, 16, 16]             128
        SkipBlock-7          [-1, 64, 16, 16]               0
           Conv2d-8          [-1, 64, 16, 16]          36,928
      BatchNorm2d-9          [-1, 64, 16, 16]             128
          Conv2d-10          [-1, 64, 16, 16]          36,928
     BatchNorm2d-11          [-1, 64, 16, 16]             128
       SkipBlock-12          [-1, 64, 16, 16]               0
          Conv2d-13          [-1, 64, 16, 16]          36,928
     BatchNorm2d-14          [-1, 64, 16, 16]             128
          Conv2d-15          [-1, 64, 16, 16]          36,928
     BatchNorm2d-16          [-1, 64, 16, 16]             128
       SkipBlock-17          [-1, 64, 16, 16]               0
          Conv2d-18          [-1, 64, 16, 16]          36,928
     BatchNorm2d-19          [-1, 64, 16, 16]             128
          Conv2d-20          [-1, 64, 16, 16]          36,928
     BatchNorm2d-21          [-1, 64, 16, 16]             128
       SkipBlock-22          [-1, 64, 16, 16]               0
          Conv2d-23          [-1, 64, 16, 16]          36,928
     BatchNorm2d-24          [-1, 64, 16, 16]             128
          Conv2d-25          [-1, 64, 16, 16]          36,928
     BatchNorm2d-26          [-1, 64, 16, 16]             128
          Conv2d-27            [-1, 64, 8, 8]           4,160
          Conv2d-28            [-1, 64, 8, 8]           4,160
       SkipBlock-29            [-1, 64, 8, 8]               0
          Conv2d-30            [-1, 64, 8, 8]          36,928
     BatchNorm2d-31            [-1, 64, 8, 8]             128
          Conv2d-32            [-1, 64, 8, 8]          36,928
     BatchNorm2d-33            [-1, 64, 8, 8]             128
       SkipBlock-34            [-1, 64, 8, 8]               0
          Conv2d-35            [-1, 64, 8, 8]          36,928
```

Purdue University 17

The Network Generated for `depth=32` (contd.)

```
  BatchNorm2d-36      [-1, 64, 8, 8]            128
      Conv2d-37       [-1, 64, 8, 8]         36,928
  BatchNorm2d-38      [-1, 64, 8, 8]            128
   SkipBlock-39       [-1, 64, 8, 8]              0
      Conv2d-40       [-1, 64, 8, 8]         36,928
  BatchNorm2d-41      [-1, 64, 8, 8]            128
      Conv2d-42       [-1, 64, 8, 8]         36,928
  BatchNorm2d-43      [-1, 64, 8, 8]            128
   SkipBlock-44       [-1, 64, 8, 8]              0
      Conv2d-45       [-1, 64, 8, 8]         36,928
  BatchNorm2d-46      [-1, 64, 8, 8]            128
      Conv2d-47       [-1, 64, 8, 8]         36,928
  BatchNorm2d-48      [-1, 64, 8, 8]            128
   SkipBlock-49       [-1, 64, 8, 8]              0
      Conv2d-50      [-1, 128, 8, 8]         73,856
  BatchNorm2d-51     [-1, 128, 8, 8]            256
      Conv2d-52      [-1, 128, 8, 8]         73,856
  BatchNorm2d-53     [-1, 128, 8, 8]            256
   SkipBlock-54      [-1, 128, 8, 8]              0
      Conv2d-55      [-1, 128, 8, 8]        147,584
  BatchNorm2d-56     [-1, 128, 8, 8]            256
      Conv2d-57      [-1, 128, 8, 8]        147,584
  BatchNorm2d-58     [-1, 128, 8, 8]            256
   SkipBlock-59      [-1, 128, 8, 8]              0
      Conv2d-60      [-1, 128, 8, 8]        147,584
  BatchNorm2d-61     [-1, 128, 8, 8]            256
      Conv2d-62      [-1, 128, 8, 8]        147,584
  BatchNorm2d-63     [-1, 128, 8, 8]            256
   SkipBlock-64      [-1, 128, 8, 8]              0
      Conv2d-65      [-1, 128, 8, 8]        147,584
  BatchNorm2d-66     [-1, 128, 8, 8]            256
      Conv2d-67      [-1, 128, 8, 8]        147,584
  BatchNorm2d-68     [-1, 128, 8, 8]            256
   SkipBlock-69      [-1, 128, 8, 8]              0
      Conv2d-70      [-1, 128, 8, 8]        147,584
  BatchNorm2d-71     [-1, 128, 8, 8]            256
      Conv2d-72      [-1, 128, 8, 8]        147,584
  BatchNorm2d-73     [-1, 128, 8, 8]            256
   SkipBlock-74      [-1, 128, 8, 8]              0
```

The Network Generated for `depth=32` (contd.)

```
      Conv2d-75        [-1, 128, 8, 8]         147,584
 BatchNorm2d-76        [-1, 128, 8, 8]             256
      Conv2d-77        [-1, 128, 8, 8]         147,584
 BatchNorm2d-78        [-1, 128, 8, 8]             256
      Conv2d-79        [-1, 128, 4, 4]          16,512
      Conv2d-80        [-1, 128, 4, 4]          16,512
   SkipBlock-81        [-1, 128, 4, 4]               0
      Conv2d-82        [-1, 128, 4, 4]         147,584
 BatchNorm2d-83        [-1, 128, 4, 4]             256
      Conv2d-84        [-1, 128, 4, 4]         147,584
 BatchNorm2d-85        [-1, 128, 4, 4]             256
   SkipBlock-86        [-1, 128, 4, 4]               0
      Conv2d-87        [-1, 128, 4, 4]         147,584
 BatchNorm2d-88        [-1, 128, 4, 4]             256
      Conv2d-89        [-1, 128, 4, 4]         147,584
 BatchNorm2d-90        [-1, 128, 4, 4]             256
   SkipBlock-91        [-1, 128, 4, 4]               0
      Conv2d-92        [-1, 128, 4, 4]         147,584
 BatchNorm2d-93        [-1, 128, 4, 4]             256
      Conv2d-94        [-1, 128, 4, 4]         147,584
 BatchNorm2d-95        [-1, 128, 4, 4]             256
   SkipBlock-96        [-1, 128, 4, 4]               0
      Conv2d-97        [-1, 128, 4, 4]         147,584
 BatchNorm2d-98        [-1, 128, 4, 4]             256
      Conv2d-99        [-1, 128, 4, 4]         147,584
BatchNorm2d-100        [-1, 128, 4, 4]             256
  SkipBlock-101        [-1, 128, 4, 4]               0
     Linear-102           [-1, 1000]       2,049,000
     Linear-103             [-1, 10]          10,010
================================================================
Total params: 5,578,498
Trainable params: 5,578,498
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 6.52
Params size (MB): 21.28
Estimated Total Size (MB): 27.82
----------------------------------------------------------------
```

**Pay attention to the total number of learnable parameters and to how
many of those are contributed to by the convolutional layers.**

Comparing the Results

- Based on the output of the script `playing_with_skip_connections.py` in the Examples directory of the `DLStudio-1.0.6` distribution.

- The `BMEnet` constructor was called as follows for the two plots:

```
for the plot in red:          BMEnet(skip_connections=True, depth=32)
for the plot in blue:         BMEnet(skip_connections=False, depth=32)
```
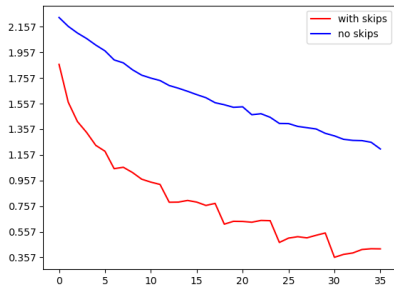
Outline

The Problem of Vanishing Gradients

- There are many publications in the literature that talk about the problem of vanishing gradients in deep networks. In my opinion, the 2010 paper "*Understanding the Difficulty of Training Deep Feedforward Neural Networks*" by Glorot and Bengio is the best. You can access it here:

  http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

- To convey the key arguments in the paper, I'll use the notation described on the next slide.

- Subsequently, I'll present Glorot and Bengio's equations for the calculation of the gradients of loss with respect to the learnable parameters in the different layers of a network.

The Notation

$\mathbf{x}$: represents the input to the neural network

L: represents the loss at the output of the final layer

$\mathbf{z}^i$: represents the input for the ith layer

z_k^i: represents the kth element of $\mathbf{z}^i$

$\mathbf{s}^i$: represents the preactivation output for the ith layer

s_i^k: represents the kth element of $\mathbf{s}^i$

$\mathbf{W}^i$: represents the link weights between the input for i^{th} layer and its preactivation output

$w_{l,k}^i$: represents the value at the index pair (l, k) of the weight $\mathbf{W}^i$

$\mathbf{b}^i$: represents the bias value needed at the output for the ith layer

$f()$: represents the activation function for the ith layer

$f'()$: represents the partial derivation of the activation function for the ith layer with respect to its argument

Equations for Backpropagating the Loss

- We start with the same relationships between the input and the output for a layer that you have seen previously in Prof. Bouman's slides:

$$\mathbf{s}^i = \mathbf{z}^i \mathbf{W}^i + \mathbf{b}^i \tag{1}$$

$$\mathbf{z}^{i+1} = f(\mathbf{s}^i) \tag{2}$$

The post-activation output for the ith layer is the input for the $(i+1)^{th}$ layer, hence the notation $\mathbf{z}^{i+1}$ in the second equation.

- Let's now take the partial derivative of $\mathbf{L}$ — which, in principle, could be any scalar — with respect to the pre-activation values s_i^k and the learnable weights $w_{l,k}^i$:

$$\frac{\partial \mathbf{L}}{\partial s_i^k} = f'(s_i^k) \, W_{k,\bullet}^{i+1} \, \frac{\partial \mathbf{L}}{\partial \mathbf{s}^{i+1}} \tag{3}$$

$$\frac{\partial \mathbf{L}}{\partial w_{l,k}^i} = z_l^i \frac{\partial \mathbf{L}}{\partial s_i^k} \tag{4}$$

In my opinion, these equations are better expressed using the Einstein notation that Prof. Bouman has previously covered in this class.

Relationship Between the Variances

- Glorot and Bengio have argued that if the weights are properly initialized, their variances can be made to remain approximately the same during forward propagation. [BTW, how to best initialize the weights is an issue unto itself in DL.] This assumption plays an important role in our examination of the gradients of the loss as they are backpropagated.

- We will start by examining the propagation of the variances in the forward direction.

- We use the following three observations: (1) The variance of a sum of n identically distributed and independent zero-mean random variables is simply n times the variance of each. (2) The variance of a product of two such random variables is the product of the two variances. And (3) In a linear chain of dependencies between such random variables, the variance at the output of the chain will be a product of the variances at the intermediate nodes.

Forward Propagating the Variances

- A variant of the first of the three observations on the last slide is that the variance of a weighted sum of n random variables of the type mentioned there equals a weighted sum of the individual variance provided you square the weights.

- These observations dictate that if we assume that all the information between the input and the output is passing through that portion of the activations where the input/output are linearly related, we can write the following approximate expression for the variance in the input to the ith layer where n_i is the size of the layer:

$$Var[z^i] = Var[x] \cdot \prod_{i'=0}^{i-1} n_{i'} Var[w^{i'}] \tag{5}$$

The notation $Var[x]$ stands for the variance in the individual elements of the input $\mathbf{x}$, $Var[z^i]$ for the variance associated with each element of the ith layer input $\mathbf{z}^i$, and $Var[w^i]$ for the variance in each element of the weight $\mathbf{W}^i$.

- Think of the variances as "signal energy" in a network.

Backpropagating the Variances of the Gradients of Loss

- The rationale that goes into writing the approximate formula shown on the previous slide for how the variances propagate in the forward direction also dictates the following relationships for a network with d layers:

$$Var\left[\frac{\partial \mathbf{L}}{\partial s^i}\right] \quad = \quad Var\left[\frac{\partial \mathbf{L}}{\partial s^d}\right] \cdot \prod_{i'=i}^{d} n_{i'+1} Var[w^{i'}] \tag{6}$$

$$Var\left[\frac{\partial \mathbf{L}}{\partial w^i}\right] \quad = \quad \prod_{i'=0}^{i-1} n_{i'} Var[w^{i'}] \cdot \prod_{i'=i}^{d-1} n_{i'+1} Var[w^{i'}] \cdot Var[x] \cdot Var\left[\frac{\partial \mathbf{L}}{\partial s^d}\right] \tag{7}$$

- Eq. (6) says that, in the layered structure of a neural network, while the variances travel forward in the manner shown on the previous slide, variances in the gradient of a scalar that exists at the last node with respect to the preactivation values in the intermediate layers travel backwards as shown by that equation.

- As to how the variances in the gradient of the output scalar vis-a-vis the weight elements depend on the variance of the gradient of the same scalar at the output is shown by Eq. (7).

Backpropagating the Variances of the Gradients of Loss (contd.)

- If we can somehow ensure (and, as mentioned earlier on Slide 23, it is possible to do so approximately with appropriate initialization of the weights) that the weight element variances $Var[w^i]$ would remain more or less the same in all the layers, our equations for the backpropagation of the gradients of the loss simplify to:

$$Var\left[\frac{\partial \mathbf{L}}{\partial s^i}\right] \quad = \quad \left[n \cdot Var[w]\right]^{d-i} \cdot Var\left[\frac{\partial \mathbf{L}}{\partial s^d}\right] \tag{8}$$

$$Var\left[\frac{\partial \mathbf{L}}{\partial w^i}\right] \quad = \quad \left[n \cdot Var[w]\right]^{d} \cdot Var[x] \cdot Var\left[\frac{\partial \mathbf{L}}{\partial s^d}\right] \tag{9}$$

- There is a huge difference between the two equations. The RHS in the first equation depends on the layer index i, where the same in the second equation is independent of i. Recall that d is the total number of layers in the network.

- As to how to interpret this dependence of Eq. (8) on the layer index depends on whether the values of $Var[w]$ are generally less than unity or greater than unity. In either case, this dependency is a source of problems in deep networks.

Backpropagating the Variances of the Gradients of Loss (contd.)

- The two equations on the last slide say that whereas the "energy" in the gradient of the loss with respect to the weight elements is **independent** of the layer index, the same in the gradient of the loss with respect to the preactivation output in each layer will become more and more muted as $d - i$ becomes larger and larger.

- **Here you have it: A theoretical explanation for the vanishing gradients of the loss.**

Outline

Why Do Skip Connections Help?

- I'll now present what has got to be the most beautiful explanation for why using skip connections helps mitigate the problem of vanishing gradients. This explanation was first presented in a 2016 paper "*Residual Networks Behave Like Ensembles of Relatively Shallow Networks*" by Veit, Wilber, and Belongie that you can download from:

  `http://papers.nips.cc/paper/6556-residual-networks-behave-like-ensembles-of-relatively-shallow-networks`

  This paper was brought to my attention about a year ago by **Bharath Comandur** who has always managed to stay one step ahead of me in our collective (meaning RVL's) understanding of deep neural networks. (It's been very disconcerting, as you can imagine!!!!)

- The main argument made in this paper is that using skip connections turns a deep network into an ensemble of relatively shallow networks. As to what that means is illustrated in the figure in the next slide.

Unraveling a Deep Network with Skip Connections

- The figure shown below, from the previously mentioned paper by Veit, Wilber, and Belongie, nicely explains the main point of that paper.

- On the left is a 3-layer network that uses skip connections based on some chosen building block. Three layers here means three of the building blocks, whatever they may be.

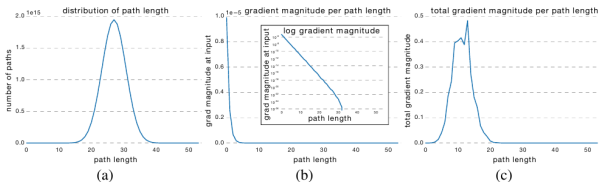- And on the right is an unraveled view of the same network.

(a) Conventional 3-block residual network   (b) Unraveled view of (a)
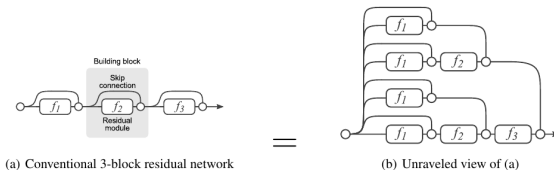
Path Lengths in the Unraveled Network

- One can argue that if there are n building blocks in a network, the total number of alternative paths in the unraveled view would be 2^n. That is because it is possible to construct a path using any desired units of the n building blocks while bypassing the others with skip connections.

- If we index the individual units in our sequence of n building blocks from 0 through $n-1$, you can also imagine each path in the unraveled view by a sequence of 1's and 0's where 1 stands for the units included and 0 for the units bypassed.

- If we think of each path as a random selection with probability p of each unit for the path, choosing i out of n for the path would constitute a binomial experiment. The outcome would be paths whose lengths are characterized by a Binomial Distribution with a mean value of $p \cdot n$. With $p = 0.5$, the average path length in our case would be $n/2$.

Only Short Paths Used for Backpropagation of the Loss Gradients

- While, the average path length may be $n/2$, it was observed experimentally by the authors that most of the loss gradients were backpropagated through very short paths.

- The paper presents a method that the authors used to measure the actual path lengths associated with the calculated gradients of the loss as they are backpropagated. The results are shown below. As the plot on the right shows, most paths are between 5 and 17 for a network with 54 units of the building block.

(a)   (b)   (c)

Outline

Visualizing the Loss Function for a Network with Skip Connections

- One could ask if it is at all possible to visualize the effect of skip connections on the shape of the loss function in the vicinity of the global minimum.

- The answer is yes and that's thanks to the 2018 paper "*Visualizing the Loss Landscape of Neural Nets*" by Li, Xu, Taylor, Studer, and Goldstein that you can access here:

  https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets.pdf

- With their visualization tool, these authors have shown that as a network becomes deeper, its loss function becomes highly chaotic. And that negatively impacts the generalization ability of a network. That is, the network will show a very small training loss and but a significant loss on the test data.

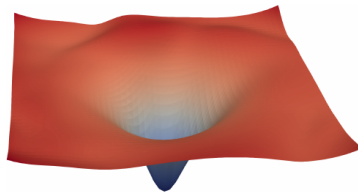# Skip Connections and the Shape of the Loss Function

- Their visualization tool also shows that when a deep network uses skip connections, the chaotic loss function becomes significantly smoother in the vicinity of the global minimum.

- They obtain their visualizations by calculating the smallest (most negative) eigenvalues of the Hessian around local minima, and visualizing the results as a heat map.

- The authors claim that, since in the vicinity of any local optimum, the loss surface is bound to be nearly convex, the paths must lie in an extremely low-dimensional space.

Visualizing the Loss Function with Skip Connections

- Shown below is the authors' visualization of the loss surface for ResNet-56 with and without skip connections.
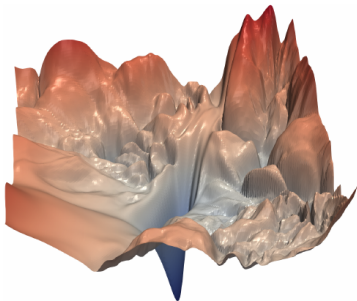
(a) without skip connections   (b) with skip connections