Using Skip Connections to Mitigate the Problem of Vanishing Gradients, and Using Batch, Instance, and Layer Normalizations for Improved SGD in Deep Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Monday 18th April, 2022 23:31

Preamble

Solving difficult machine learning problems frequently requires using deep networks.

But training deep networks can be difficult because of the vanishing gradients problem. Vanishing gradients means that the gradients of the backpropagated loss with respect to the learnable parameters 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.

Another reason for the vanishing gradient phenomenon is the saturating effect of the activation function nonlinearities.

One of the main strategies used in modern neural networks for addressing the the problem of vanishing gradients is Skip Connections.

Preamble (contd.)

With skip connections, a network includes shortcut pathways so that the loss calculated at the output can be "felt" more strongly in the earlier layers of the network during backpropagation.

That using such shortcuts allowed for deeper networks to be constructed and, thus, allowed for more difficult learning problems to be solved was first demonstrated by He, Zhang, Ren, and Sun in the following paper that proposed the **ResNet** network that won several tough image classification, object detection, and segmentation competitions in 2015:

`https://arxiv.org/pdf/1512.03385.pdf`

Another strategy that mitigates the problems caused by vanishing gradients is the in-transit data normalization *prior to invoking the nonlinear activation* in the layers of a neural network. Of the different data normalization strategies available today, Batch Normalization (BN) is arguably the most popular.

Preamble (contd.)

The effectiveness of BN for mitigating against vanishing gradients can be rationalized thus: During forward propagation, as the data flows through a deep network, the saturating property of the activation-function nonlinearities can significantly alter the statistical attributes of the data in a way that contributes to the problem of vanishing gradients. If the operating points for calculating the partial derivatives during the forward propagation of the data are in the saturating portions of the activation nonlinearities, the derivatives would acquire negligibly small values and the gradients of the backpropagated loss would also become correspondingly small.

In the following celebrated paper, Ioffe and Szegedy referred to this shift in the data properties as **Internal Covariate Shift (ICS)** and proposed the Batch Normalization (BN) algorithm for an efficient way to counter it:

`https://arxiv.org/abs/1502.03167`

Preamble (contd.)

BN normalizes the channel based values of the mean and the standard deviation of the training data as it is coursing through the network, but only to the extent made necessary by the minimization of loss at the output of the network.

As it turns out, BN has other benefits also and these include:

- BN can considerably speed up convergence of SGD, especially for very large datasets that may otherwise require distributed implementations of SGD utilizing multiple machines, with each assigned a subset of the overall training dataset. As you can imagine, distributed learning creates its own issues about how to best combine the values of the learnable parameters from the different machines.
- In addition to shortening the training time, the intra-batch normalization carried out by BN acts as a regularizer for improving the quality of the SGD based parameter updates.

Preamble (contd.)

While BN can significantly improve the quality of a neural network, it is NOT ALWAYS the best normalization strategy for the data flowing through a network. For example, the fact that BN normalizes over all of the training samples in a batch, makes it inappropriate for cases where intra-batch interference is not acceptable. That would be the case for what are known as style-transfer neural networks. The Instance Normalization (IN) algorithm is more appropriate in such cases for stabilizing the learning process and achieving faster SGD convergence.

As it turns out, neither BN nor IN is suitable for an important class of neural networks known as Recurrent Neural Networks (RNN) that I'll be talking about later in the semester for sequence-to-sequence learning. For such networks, you need what's known as Layer Normalization (LN). This lecture has separate sections that deal with each of these three types of data normalizations for the in-transit data flowing through a neural network.

Preamble (contd.)

I'll start the lecture by introducing my `SkipBlock` that was inspired by the logic of ResNet. I have used `SkipBlock` as a building block for several neural networks you will find in DLStudio.

As you will see, my definition of `SkipBlock` makes it easier to use such serially-connected building blocks for creating a deep network. For this lecture, I'll demonstrate the utility of the `SkipBlock` building block in a network I have named the `BMENet` for reasons that I'll mention in the main part of this lecture.

Subsequently, in three separate sections, I will present the algorithms that are used for BN, IN, and LN.

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

Outline

Outline

Skip Connections are an Architectural Issue

- As with any architecture, the best way to get started with learning about a new architectural feature is by looking at some examples of that feature and, in an engineering context, some evidence of its usefulness.

- What that in mind, the best starting point for you to learn about the whys and the hows of skip connections in neural networks would be to play with the following script in the Examples directory of the DLStudio module:

```
playing_with_skip_connections.py
```

- Execute the above script, see its results with and without the skip-connections feature turned on, compare the results you get, and then work your way backwards into the relevant section of the DLStudio code to get a first sense of what is accomplished by connection skipping in a neural network.

Skip Connections are an Architectural Issue (contd.)

- You can turn the skip-connections on and off in the script `playing_with_skip_connections.py` by commenting out one of the following two statements in that script:

```
model = skip_con.BMEnet(skip_connections=True, depth=32)     ## if you want skip connections

model = skip_con.BMEnet(skip_connections=False, depth=32)    ## if you don't want skip connections
```

- You can locate all of the code related to connection skipping in DLStudio by searching for the string "SkipConnections" in the module file. That is the name of the demo class for experimenting with skip connections in DLStudio.

- The demo class `SkipConnections` in DLStudio contains a network named `BMEnet` that is constructed using the building-block class `SkipBlock`.

Skip Connections are an Architectural Issue (contd.)

- **I have used "BME" in the name `BMEnet` network in honor of the School of Biomedical Engineering, Purdue University.** Without their sponsorship, you would not be taking BME646/ECE695DL this semester.

- In what follows, I'll first present `SkipBlock` to demonstrate how one can define what are known as building-block network elements. The idea here is to define a small network with more than one pathway from its input to the output.

- Then I'll present the `BMEnet` as a network constructed using the building-block elements `SkipBlock`.

- I'll follow that with a presentation of classification results with and without connection skipping in `BMEnet`.

Outline

SkipBlock as a Building-Block Network in DLStudio

- Just as `torch.nn.Conv2d`, `torch.nn.Linear`, etc., are the building blocks of a general CNN, `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.

Two Assumptions Implicit in the Definition of SkipBlock

- As shown on the next slide, an instance of `SkipBlock` consists of two convolutional layers, as you can see in lines (E) and (F). Each convolutional output is subject to batch normalization and ReLU activation. Also note the additional convo layer provided by the downsampler in line (J) when the Boolean var `downsampler` is `True`. The design of `SkipBlock` is based on the following two assumptions:
  1. 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 (L) in identity with the output in lines (Y) through (c).

```
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                                          ## (A)
        self.skip_connections = skip_connections                              ## (B)
        self.in_ch = in_ch                                                    ## (C)
        self.out_ch = out_ch                                                  ## (D)
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)        ## (E)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)        ## (F)
        self.bn1 = nn.BatchNorm2d(out_ch)                                     ## (G)
        self.bn2 = nn.BatchNorm2d(out_ch)                                     ## (H)
        if downsample:                                                        ## (I)
            ## setting the stride=2 halves the size of the image:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)          ## (J)

    def forward(self, x):                                                     ## (K)
        identity = x                                                          ## (L)
        out = self.convo1(x)                                                  ## (M)
        out = self.bn1(out)                                                   ## (N)
        out = torch.nn.functional.relu(out)                                   ## (O)
        if self.in_ch == self.out_ch:                                         ## (P)
            out = self.convo2(out)                                            ## (Q)
            out = self.bn2(out)                                               ## (R)
            out = torch.nn.functional.relu(out)                               ## (S)
```


Continued on the next slide ....

SkipBlock's Definition (contd.)

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

```
if self.downsample:                                          ## (T)
    out = self.downsampler(out)                              ## (U)
    identity = self.downsampler(identity)                    ## (V)
if self.skip_connections:                                    ## (W)
    if self.in_ch == self.out_ch:                            ## (X)
        out += identity                                      ## (Y)
    else:                                                    ## (Z)
        out[:,:self.in_ch,:,:] += identity                   ## (a)
        out[:,self.in_ch:,:,:] += identity                   ## (b)
return out                                                   ## (c)
```

- The SkipBlock design allows you to create the following different types of instance of the building block:
  - Instances with the same number of input/output channels and the same input/output image sizes. These would be useful for providing depth with shortcuts in a neural network.
  - Instances meant primarily for creating a resolution hierarchy by progressively halving the image size with each skip block.
  - Instances that downsample the input image to half its size while doubling the number of input channels at the output.

SkipBlock's Definition (contd.)

- Creating shortcuts from the input to the output of a building-block module like SkipBlock must contend with the issues that arise when the number of channels at the input and the output are not the same and also when the input/output images sizes are meant to be different.

- When the number of channels is the same at the input and the output, we can straightforwardly mix the input and the output as in line (Y) shown on the previous slide.

- However, when that is not the case, we use the mixing logic in lines (a) and (b) on the previous slide. Recall that SkipBlock is based on the assumption that when the input/output channels are different, you have twice the channels in the output than in the input. Line (a) mixes the input with the upper half of the channels in the output and line (b) does the same with the lower half of the output channels.

Outline

The BMEnet Class — Building a Network with SkipBlock

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

```
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]:                                                    ## (A)
            sys.exit("BMEnet has been tested for depth for only 16, 32, and 64")
        self.depth = depth // 8                                                             ## (B)
        self.conv = nn.Conv2d(3, 64, 3, padding=1)                                          ## (C)
        self.pool = nn.MaxPool2d(2, 2)              ## keep max over 2x2 and shift by 2    ## (D)
        self.skip64_arr = nn.ModuleList()                                                   ## (E)
        for i in range(self.depth):                                                         ## (F)
            self.skip64_arr.append(DLStudio.SkipConnections.SkipBlock(64, 64,
                                                    skip_connections=skip_connections))     ## (G)
        self.skip64ds = DLStudio.SkipConnections.SkipBlock(64, 64, downsample=True,
                                                     skip_connections=skip_connections)     ## (H)
        self.skip64to128 = DLStudio.SkipConnections.SkipBlock(64, 128,
                                                    skip_connections=skip_connections )     ## (I)
        self.skip128_arr = nn.ModuleList()                                                  ## (J)
        for i in range(self.depth):                                                         ## (K)
            self.skip128_arr.append(DLStudio.SkipConnections.SkipBlock(128, 128,
                                                    skip_connections=skip_connections))     ## (L)
        self.skip128ds = DLStudio.SkipConnections.SkipBlock(128,128, downsample=True,
                                                     skip_connections=skip_connections)     ## (M)
        ## The following declaration works only for the cases when the product (CxHxW) for the
        ## topmost convo layer equals 2048:
        self.fc1 =  nn.Linear(2048, 1000)                                                   ## (N)
        self.fc2 =  nn.Linear(1000, 10)                                                     ## (O)
```


Continued on the next slide ....

The BMEnet Class (contd.)

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

```
def forward(self, x):                                                          ## (P)
    ##   The following call will halve the size of the input image because you are
    ##   maxing over 2x2 windows and shifting the window by 2 pixels
    x = self.pool(torch.nn.functional.relu(self.conv(x)))                      ## (Q)
    for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):                ## (R)
        x = skip64(x)                                                          ## (S)
    x = self.skip64ds(x)                                                       ## (T)
    for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):                ## (U)
        x = skip64(x)                                                          ## (V)
    x = self.skip64ds(x)                                                       ## (W)
    x = self.skip64to128(x)                                                    ## (X)
    for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):              ## (Y)
        x = skip128(x)                                                         ## (Z)
    ##   If you uncomment the statement that follows, you will get an error because of the
    ##   comment I have made just above line (N) --- assuming you are working with the 32x32
    ##   CIFAR10 images.
    ## x = self.skip128ds(x)                                                   ## (a)
    for i,skip128 in enumerate(self.skip128_arr[self.depth//4:]):              ## (b)
        x = skip128(x)                                                         ## (c)
    x = x.view(-1, 2048 )                                                      ## (d)
    x = torch.nn.functional.relu(self.fc1(x))                                  ## (e)
    x = self.fc2(x)                                                            ## (f)
    return x                                                                   ## (g)
```

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 BMEnet Network Used for Comparison Results

- 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 couple of slides shows the details of the network constructed when the constructor option depth is set to 32.

- The network summary shown on the next couple of slides was produced by the Python module torch-summary and **NOT** torchsummary. [**NOTE:** The former give superior results as far as I can tell. What adds to the confusion about these two modules is that while they are both available at "https://pypi.org" under their respective names, they are both imported with the same import statement such as "from torchsummary import summary".]

- At the end of the network detail, you will see that the total number of learnable parameters is around 3.6 million, only half of which are contributed to by the convolutional layers, with the rest coming from the fully connected layers at the top of the network.

The Network Generated for depth=32

```
a summary of input/output for the model:
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
|-Conv2d: 1-1                            [-1, 64, 32, 32]          1,792
|-MaxPool2d: 1-2                         [-1, 64, 16, 16]          --
|-ModuleList: 1                          []                        --
|    |--SkipBlock: 2-1                    [-1, 64, 16, 16]          --
|    |    |-Conv2d: 3-1                   [-1, 64, 16, 16]          36,928
|    |    |-BatchNorm2d: 3-2              [-1, 64, 16, 16]          128
|    |    |-Conv2d: 3-3                   [-1, 64, 16, 16]          36,928
|    |    |-BatchNorm2d: 3-4              [-1, 64, 16, 16]          128
|-SkipBlock: 1-3                         [-1, 64, 8, 8]            --
|    |-Conv2d: 2-2                       [-1, 64, 16, 16]          36,928
|    |-BatchNorm2d: 2-3                  [-1, 64, 16, 16]          128
|    |-Conv2d: 2-4                       [-1, 64, 16, 16]          36,928
|    |-BatchNorm2d: 2-5                  [-1, 64, 16, 16]          128
|    |-Conv2d: 2-6                       [-1, 64, 8, 8]            4,160
|    |-Conv2d: 2-7                       [-1, 64, 8, 8]            (recursive)
|-ModuleList: 1                          []                        --
|    |-SkipBlock: 2-8                    [-1, 64, 8, 8]            --
|    |    |-Conv2d: 3-5                   [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-6              [-1, 64, 8, 8]            128
|    |    |-Conv2d: 3-7                   [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-8              [-1, 64, 8, 8]            128
|    |-SkipBlock: 2-9                    [-1, 64, 8, 8]            --
|    |    |-Conv2d: 3-9                   [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-10             [-1, 64, 8, 8]            128
|    |    |-Conv2d: 3-11                  [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-12             [-1, 64, 8, 8]            128
|    |-SkipBlock: 2-10                   [-1, 64, 8, 8]            --
|    |    |-Conv2d: 3-13                  [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-14             [-1, 64, 8, 8]            128
|    |    |-Conv2d: 3-15                  [-1, 64, 8, 8]            36,928
|    |    |-BatchNorm2d: 3-16             [-1, 64, 8, 8]            128
|-SkipBlock: 1-4                         [-1, 64, 4, 4]            (recursive)
|    |-Conv2d: 2-11                      [-1, 64, 8, 8]            (recursive)
|    |-BatchNorm2d: 2-12                 [-1, 64, 8, 8]            (recursive)
|    |-Conv2d: 2-13                      [-1, 64, 8, 8]            (recursive)
|    |-BatchNorm2d: 2-14                 [-1, 64, 8, 8]            (recursive)
|    |-Conv2d: 2-15                      [-1, 64, 4, 4]            (recursive)
|    |-Conv2d: 2-16                      [-1, 64, 4, 4]            (recursive)
```


Continued on the next slide ....

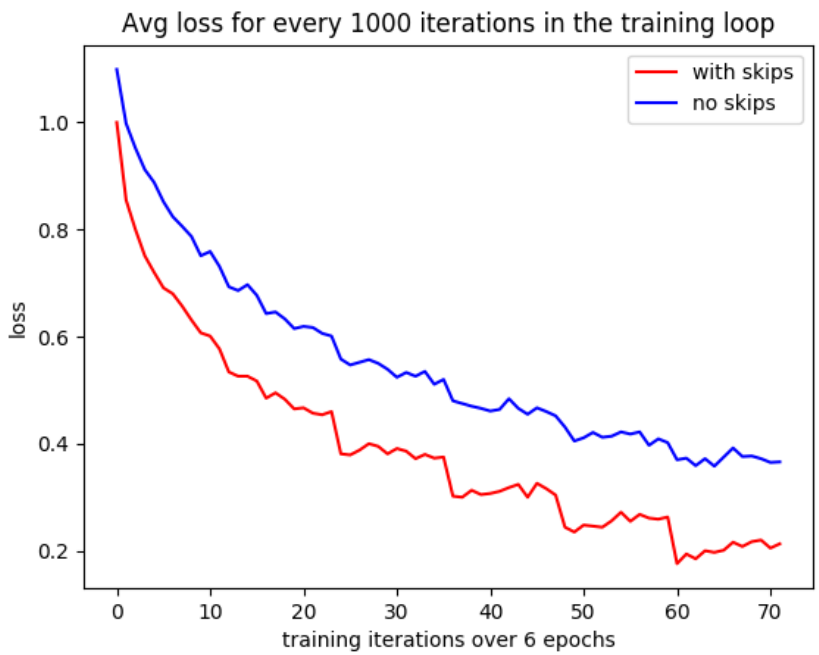
The Network Generated for `depth=32` (contd.)

```
|-SkipBlock: 1-5                        [-1, 128, 4, 4]           --
|    |-Conv2d: 2-17                     [-1, 128, 4, 4]           73,856
|    |-BatchNorm2d: 2-18                [-1, 128, 4, 4]           256
|-ModuleList: 1                         []                        --
|    |-SkipBlock: 2-19                  [-1, 128, 4, 4]           --
|    |    |-Conv2d: 3-17                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-18           [-1, 128, 4, 4]           256
|    |    |-Conv2d: 3-19                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-20           [-1, 128, 4, 4]           256
|    |_SkipBlock: 2-20                  [-1, 128, 4, 4]           --
|    |    |-Conv2d: 3-21                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-22           [-1, 128, 4, 4]           256
|    |    |-Conv2d: 3-23                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-24           [-1, 128, 4, 4]           256
|    |-SkipBlock: 2-21                  [-1, 128, 4, 4]           --
|    |    |-Conv2d: 3-25                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-26           [-1, 128, 4, 4]           256
|    |    |-Conv2d: 3-27                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-28           [-1, 128, 4, 4]           256
|    |-SkipBlock: 2-22                  [-1, 128, 4, 4]           --
|    |    |-Conv2d: 3-29                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-30           [-1, 128, 4, 4]           256
|    |    |-Conv2d: 3-31                [-1, 128, 4, 4]           147,584
|    |    |-BatchNorm2d: 3-32           [-1, 128, 4, 4]           256
|-Linear: 1-6                           [-1, 1000]                2,049,000
|-Linear: 1-7                           [-1, 10]                  10,010
==========================================================================================
Total params: 3,692,354
Trainable params: 3,692,354
Non-trainable params: 0
Total mult-adds (M): 82.94
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 2.20
Params size (MB): 14.09
Estimated Total Size (MB): 16.29
==========================================================================================
The number of learnable parameters in the model: 4078658
The number of layers in the model: 98
```

Comparing the Results

- The results were produced by the script `playing_with_skip_connections.py` in the `Examples` directory of the DLStudio module with 6 epochs of training.
- The constructor for the `BMEnet` network 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)
```



Comparing the Results (contd.)

- For the case when the skip connections are **NOT** used, shown below are the classification results on the unseen test data (10,000 samples) after 6 epochs of training.
- Note in particular the significant "confusion" between the "cat" and the "dog" categories. Over 17% of the cats in the test data are labeled as dogs and over 14% of the dogs labeled as cats.

```
Prediction accuracy for plane : 83 %
Prediction accuracy for   car : 91 %
Prediction accuracy for  bird : 58 %
Prediction accuracy for   cat : 49 %
Prediction accuracy for  deer : 65 %
Prediction accuracy for   dog : 63 %
Prediction accuracy for  frog : 86 %
Prediction accuracy for horse : 71 %
Prediction accuracy for  ship : 84 %
Prediction accuracy for truck : 75 %

Overall accuracy of the network on the 10000 test images: 73 %

Displaying the confusion matrix:

           plane    car   bird    cat   deer    dog   frog  horse   ship  truck

 plane:    83.70   2.20   1.70   1.20   0.50   0.30   1.00   1.10   6.70   1.60
   car:     1.40  91.50   0.00   0.10   0.10   0.10   1.00   0.00   2.60   3.20
  bird:    13.20   1.20  58.70   4.60   5.00   5.50   9.00   1.20   1.20   0.40
   cat:     4.30   1.90   7.20  49.40   4.90  17.30   9.20   1.90   3.30   0.60
  deer:     4.60   0.60   9.80   5.60  65.90   2.60   6.90   2.90   0.90   0.20
   dog:     2.50   0.80   6.40  14.30   3.70  63.90   2.40   3.70   1.70   0.60
  frog:     0.60   0.40   3.10   4.30   1.40   1.20  86.50   0.40   1.80   0.30
 horse:     4.30   0.60   2.90   4.50   7.00   6.80   1.30  71.00   0.40   1.20
  ship:     8.10   3.10   0.60   0.80   0.50   0.50   0.70   0.00  84.40   1.30
 truck:     4.50  11.60   0.40   1.50   0.10   0.40   1.10   1.40   3.70  75.30
```

Comparing the Results (contd.)

- And here are the results for the case when skip connections are used.
- While the overall classification performance has gone up by 2%, there is significant improvement (almost 10%) for the "cat" category. However, the performance for the "dog" category has taken a 5% hit. But note we are talking about only 6 epochs of training.

```
Prediction accuracy for plane : 79 %
Prediction accuracy for   car : 88 %
Prediction accuracy for  bird : 63 %
Prediction accuracy for   cat : 58 %
Prediction accuracy for  deer : 77 %
Prediction accuracy for   dog : 58 %
Prediction accuracy for  frog : 89 %
Prediction accuracy for horse : 75 %
Prediction accuracy for  ship : 89 %
Prediction accuracy for truck : 73 %

Overall accuracy of the network on the 10000 test images: 75 %

Displaying the confusion matrix:

           plane    car   bird    cat   deer    dog   frog  horse   ship  truck

 plane:    79.60   1.10   2.50   1.00   2.40   0.60   1.30   0.70   9.10   1.70
   car:     0.90  88.40   0.20   0.30   0.50   0.00   1.00   0.00   4.00   4.70
  bird:     6.70   0.50  63.80   4.10   7.70   3.70   9.00   2.10   2.00   0.40
   cat:     1.80   1.00   5.10  58.90   8.40  10.00   9.70   2.00   2.30   0.80
  deer:     1.30   0.30   5.60   3.90  77.20   1.50   6.20   2.30   1.70   0.00
   dog:     1.40   0.70   4.50  17.30   6.50  58.70   5.20   3.90   1.20   0.60
  frog:     0.50   0.00   2.70   2.80   2.00   0.70  89.80   0.20   1.30   0.00
 horse:     0.60   0.40   3.00   4.30   8.70   3.80   1.90  75.70   1.10   0.50
  ship:     4.20   2.30   0.50   0.90   0.90   0.20   1.10   0.10  89.10   0.70
 truck:     3.70  13.20   0.50   1.20   1.00   0.40   1.10   1.10   4.20  73.60
```

What? All That Work for Just 2% Improvement!

- Regarding the importance of connection skipping, you don't want to be fooled by the just 2% improvement in the classification accuracy shown on the previous slide.
  [The network used in that example is very simple and the number of epochs used for training not that many. That example was used just to introduce you to the idea of connection skipping.]

- In the more complex neural networks you will see during the rest of this semester, connection skipping can make the difference between getting no results at all and getting a good performance.

- For example, the Encoder-Decoder architectures for semantic segmentation of images would not work at all without the shortcuts made possible by connection skipping. The shortcuts in these networks are from the encoder side to the decoder side. The Encoder-Decoder networks are presented in Week 9.

- Such shortcuts are also required for the stability of learning and for improving the convergence in GANs (Generative Adversarial Networks, Week 11) and Transformers that I'll cover in Week 14.

Outline

Batch Normalization

- In the Preamble, I mentioned the BN algorithm proposed by Ioffe and Szegedy. This algorithm is now universally used in neural networks for batch normalization. The reason for its popularity is that its computational overhead is tiny — especially in relation to its benefits. All it requires is the learning of two additional scalar parameters per channel in a network layer. The two parameters, known as the *learnable affine parameters for a channel*, control the extent to which data normalization is carried out in that channel of a layer.

- For explaining BN, I like the notation that I first saw in the following paper by Huang and Belongie [It's a great paper worth reading particularly if you are interested in using deep networks for style transfer. Style transfer generally means slapping the image texture from one class of images onto a different class of images.]

`https://arxiv.org/pdf/1703.06868.pdf`

Batch Normalization (contd.)

- We will use our usual notation (B, C, H, W) to represent a batch of images at the pre-activation point in a layer in a network. As you'll recall, B is the number of images in the batch, C the number of output channels produced by the layer, and $H \times W$ the height and the width of the pixel arrays in the batch.

- I'll use the notation x_{nchw} to represent the value of the pixel at the coordinates (h, w) in the channel indexed c in the batch image indexed n.

- BN computes the mean μ_c and the standard-deviation σ_c of the pixel values separately in each channel indexed c:

$$\mu_c = \frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} x_{nchw} \tag{1}$$

$$\sigma_c = \sqrt{\frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} (x_{nchw} - \mu_c)^2 + \epsilon} \tag{2}$$

Batch Normalization (contd.)

- In the last formula shown on the previous slide, a small number ϵ is added to what you get after the triple summation for the averaging of the inner square. We need this ϵ to protect against division by zero in the formula shown below for the case when the data in a channel is all constants.

- Batch normalization means replacing every pixel value x_{nchw} in a channel c by the following value:

$$BN(x_{nchw}) \quad = \quad \gamma_c \frac{x_{nchw} \; - \; \mu_c}{\sigma_c} \; + \; \beta_c \tag{3}$$

- The parameters γ_c and β_c are referred to as the scale and the shift parameters for the channel in question. More generally, they are together called the the affine parameters that must be learned on a per channel basis in each layer that uses BN.

Batch Normalization (contd.)

- To reiterate, the goal of BN is to normalize the mean and standard deviation for each individual feature channel — but only to the extent permitted by the learnable affine parameters γ and β.

- All of our discussion so far in this section has related to BN as you would use it during the training of a network. That leads to the following question: Does the idea of batch normalization also apply at the inference time?

- Obviously, since the notion of a batch does not really exist at the inference time, thinking of BN for inference sounds silly. [One certainly has the option of inferencing with batches for generating the results faster, but there is no SGD going on at inference time and therefore no need for batch based processing of data.]

- And that, in turn, leads to the following question: If BN is fundamentally a training-time idea, what do we do with the affine parameters (γ_c, β_c) during inference?

Population Stats versus Batch Stats

- Before getting to the questions posed on the previous slide, note that, in addition to computing the per-channel value for the mean and the standard-deviation as previously explained, PyTorch also computes the per-channel values for what are known as the population mean and the population standard-deviation.
- During the training phase, the populations stats are calculated from the batch-based stats (μ_c, σ_c) using the following recursive update formulas in which θ is a user-specified decay parameter that is typically set to something like 0.99.

$$pop_{mean,c} = pop_{mean,c} * \theta + \mu_c * (1 - \theta) \quad (4)$$

$$popu_{sd,c} = pop_{sd,c} * \theta + \sigma_c * (1 - \theta) \quad (5)$$

- The recursion starts by setting $pop_{mean,c}$ to the first value encountered for μ_c and $pop_{sd,c}$ to the first value encountered for σ_c, both for the channel c in question.

Population Stats versus Batch Stats (contd.)

- The work that is done by (μ_c, σ_c) at the training time is accomplished by the population mean and the population standard deviation at the inference time **using the same affine parameters that were learned during training**.
- As to how exactly the population stats are used at inference time depends on whether or not you have put the trained model in what's known as the **eval** mode. If *model* denotes the network after it's been trained, the following invocation places the network in the eval model.

```
model = model.eval()
```

- If you test your trained network in eval mode, the population stats are frozen at the values computed during training. **However, if you do not use the eval mode, the population stats would continue to be updated even during testing.**
- The great researchers I work with in RVL tell me that you get the best results at testing time if you use your trained model in the eval mode.

Outline

Instance Normalization

- Instance Normalization is important for those applications of DL where you cannot tolerate intra-batch interference in the calculation of loss. A prime example of this is the networks for style transfer. When style loss is calculated with BN layers in a network, the calculated loss may be a poor approximation to the input-image to target-image style difference. Replacing the BN layers with IN layers is a solution to this problem.

- With regard to the calculations involved, the only difference between BN and IN is that for the latter the averaging that you saw in Eqs. (1) and (2) is now carried out on a *per-channel and per-instance* basis as opposite to on a per-channel basis.

Instance Normalization (contd.)

- For the formulas for IN, as before, the index c is for the channel and the index n for the instance in a batch:

$$\mu_{cn} = \frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W} x_{nchw} \tag{6}$$

$$\sigma_{cn} = \sqrt{\frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W}(x_{nchw} - \mu_{cn})^2 + \epsilon} \tag{7}$$

- These formulas imply that in each output channel you have to keep track of the mean and the standard deviation for each instance index in a batch separately.

- Instance normalization means replacing every pixel value x_{nchw} in a channel c of the instance n in a batch by the following value:

$$IN(x_{nchw}) = \gamma_{cn}\frac{x_{nchw} - \mu_{cn}}{\sigma_{cn}} + \beta_{cn} \tag{8}$$

Instance Normalization (contd.)

- The parameters γ_{cn} and β_{cn} in the formulas on the previous slide are now the scale and the shift parameters for the channel in question. These parameters must be kept track of on a per-instance basis in each channel.

- A most noteworthy aspect of IN is that now you can use the same formula for both training and testing. After you have learned the Affine Parameters γ_{cn} and β_{cn} for each channel on a per-instance basis, at inference time, you use the same formulas as shown on the previous slide for calculating the mean and the variance for the test image in question. Recall, inferencing will generally involve only one instance at a time and not a batch of instances.

Outline

Layer Normalization (LN)

- As I mentioned in the Preamble, neither BN nor IN is appropriate for an important class of neural networks known as Recurrent Neural Networks (RNN).
- As to how exactly the data mean and the standard-deviation, (μ, σ), are calculated for LN depends on whether or not your application can tolerate intra-batch interference. In general, you have the following two options for the case of image data:
  - OPTION 1: For each instance in a batch, you calculate the (μ,σ) stats using the data *in all the channels*. [For both BN and IN, each channel was dealt with separately for calculating (μ,σ) values]
  - OPTION 2: You use **all** the data in a layer for estimating (μ,σ). That is, your calculation now includes *all of the batch instances and all the channels* for each instance.

Layer Normalization (contd.)

- For numerical sequence data in time-series data prediction applications, you are likely to use the first option on the previous slide.
- On the other hand, for symbolic sequence data as in language processing (see my Week 13 lecture), your (μ,σ) estimation is likely to be based on the embedding vectors associated with the words. This would be similar to the second option on the previous slide. That is, you would be using *all* of the data in a layer for calculing the (μ,σ) values.
- Shown on the next slide are the formulas for the first option for image data applications. In the formulas for calculating μ and σ shown earlier in Eqs. (1) and (2) on Slide 34, we now include the pixel values in all of the channels but do so for each batch instance separately. Recall that in BN on Slide 34, we calculated the stats on a per-channel basis across all the batch instances. But now we are going to compute them on an instance basis but over all the channels.

Layer Normalization (contd.)

- Here are the LN equations for calculating the (μ,σ) values for image data (as based on Option 1):

$$\mu_n = \frac{1}{CHW}\sum_{c=1}^{C}\sum_{h=1}^{H}\sum_{w=1}^{W} x_{nchw} \tag{9}$$

$$\sigma_n = \sqrt{\frac{1}{CHW}\sum_{c=1}^{C}\sum_{h=1}^{H}\sum_{w=1}^{W} (x_{nchw} - \mu_n)^2 + \epsilon} \tag{10}$$

- In these formulas, as you'll recall, B is the batch size, C the number of channels in the layer, H the height of the image, and W the width and n the instance index in a batch.
- LN means replacing every pixel value x_{nchw} by the following value:

$$LN(x_{nchw}) = \gamma_n \frac{x_{nchw} - \mu_n}{\sigma_n} + \beta_n \tag{11}$$

- The affine parameters γ_n and β_n are now the batch-instance-specific scale and the shift parameters.

Layer Normalization (contd.)

- By the way, for *all three cases* of data normalization, the two parameters γ and β are also referred to as the *gain* and *bias* parameters that must be learned.
- Shown below are the formulas for calculating the (μ, σ) values for the second option mentioned on Slide 44:

$$\mu_l = \frac{1}{BCHW} \sum_{n=1}^{B} \sum_{c=1}^{C} \sum_{h=1}^{H} \sum_{w=1}^{W} x_{nchw} \tag{12}$$

$$\sigma_l = \sqrt{\frac{1}{BCHW} \sum_{n=1}^{B} \sum_{c=1}^{C} \sum_{h=1}^{H} \sum_{w=1}^{W} (x_{nchw} - \mu_l)^2 + \epsilon} \tag{13}$$

where l is the layer index. Now LN means replacing every pixel value x_{nchw} by the following value:

$$LN(x_{nchw}) = \gamma_l \frac{x_{nchw} - \mu_l}{\sigma_l} + \beta_l \tag{14}$$

- Now only the two affine parameters γ_l and β_l need to be learned for each layer.

Layer Normalization (contd.)

- PyTorch gives you a parameter `normalized_shape` for the constructor of the class `torch.nn.LayerNorm` that you can use to choose between the different ways of implementing Layer Normalization.

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

- x: represents the input to the neural network
- L: represents the loss at the output of the final layer
- z^i: represents the input for the i^{th} layer
- z_k^i: represents the k^{th} element of z^i
- s^i: represents the preactivation output for the i^{th} layer
- s_i^k: represents the k^{th} element of s^i
- 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 W^i
- b^i: represents the bias value needed at the output for the i^{th} layer
- $f()$: represents the activation function for the i^{th} layer
- $f'()$: represents the partial derivation of the activation function for the i^{th} 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 saw in the last section of my Week 3 slides:

$$\mathsf{s}^i = \mathsf{z}^i \mathsf{W}^i + \mathsf{b}^i \tag{15}$$

$$\mathsf{z}^{i+1} = f(\mathsf{s}^i) \tag{16}$$

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

- Let's now take the partial derivative of L with respect to the pre-activation values s_i^k and the learnable weights $w_{l,k}^i$ [These are similar to what you saw in the last section of Week 3 slides. The difference is with respect to where the activation function is taken into account.]:

$$\frac{\partial \mathsf{L}}{\partial s_i^k} = f'(s_i^k)\, W_{k,\bullet}^{i+1}\, \frac{\partial \mathsf{L}}{\partial \mathsf{s}^{i+1}} \tag{17}$$

$$\frac{\partial \mathsf{L}}{\partial w_{l,k}^i} = z_l^i \frac{\partial \mathsf{L}}{\partial s_i^k} \tag{18}$$

  where the "filled circle" you see in $W_{k,\bullet}^{i+1}$ means that you are decimating the second index for the elements of the matrix — which is what happens in the product of a matrix with a column vector.

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 i^{th} layer where n_i is the size of the layer:

$$Var[z^i] \;=\; Var[x] \cdot \prod_{i'=0}^{i-1} n_{i'}\, Var[w^{i'}] \tag{19}$$

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

$$Var\left[\frac{\partial \mathrm{L}}{\partial s^i}\right] = Var\left[\frac{\partial \mathrm{L}}{\partial s^d}\right] \cdot \prod_{i'=i}^{d} n_{i'+1} Var[w^{i'}] \tag{20}$$

$$Var\left[\frac{\partial \mathrm{L}}{\partial w^i}\right] = \prod_{i'=0}^{i-1} n_{i'} Var[w^{i'}] \cdot \prod_{i'=i}^{d-1} n_{i'+1} Var[w^{i'}] \cdot Var[x] \cdot Var\left[\frac{\partial \mathrm{L}}{\partial s^d}\right] \tag{21}$$

- Eq. (20) 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. (21).

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

- If we can somehow ensure (and, as mentioned earlier on Slide click, 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:

$$Var\left[\frac{\partial \mathrm{L}}{\partial s^i}\right] = \Big[n \cdot Var[w]\Big]^{d-i} \cdot Var\left[\frac{\partial \mathrm{L}}{\partial s^d}\right] \tag{22}$$

$$Var\left[\frac{\partial \mathrm{L}}{\partial w^i}\right] = \Big[n \cdot Var[w]\Big]^{d} \cdot Var[x] \cdot Var\left[\frac{\partial \mathrm{L}}{\partial s^d}\right] \tag{23}$$

- 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. (22) 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 previous 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.
- Eq. (22) speaks to the backprojection of the prediction error from the final output to i^{th} layer and Eq. (23) to the calculation of the gradient of the loss with respect to the learnable parameters in the same layer. On the face of it, it seems that only the backprojection of the prediction errors is affected by the phenomenon of the vanishing gradients. But note that the greater the errors in the backproped prediction errors, the more meaningless the updates calculated for the learnable parameters in that layer.
- **Now 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:

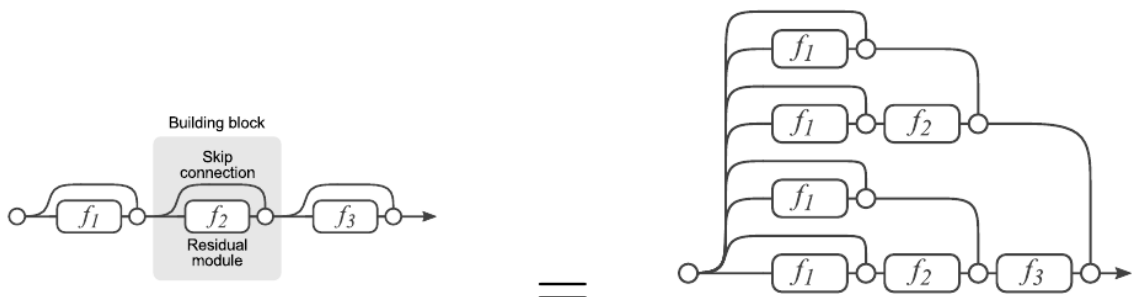
  `http://papers.nips.cc/paper/6556-residual-networks-behave-like-ensembles-of-relatively-shallow-networks`

  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.



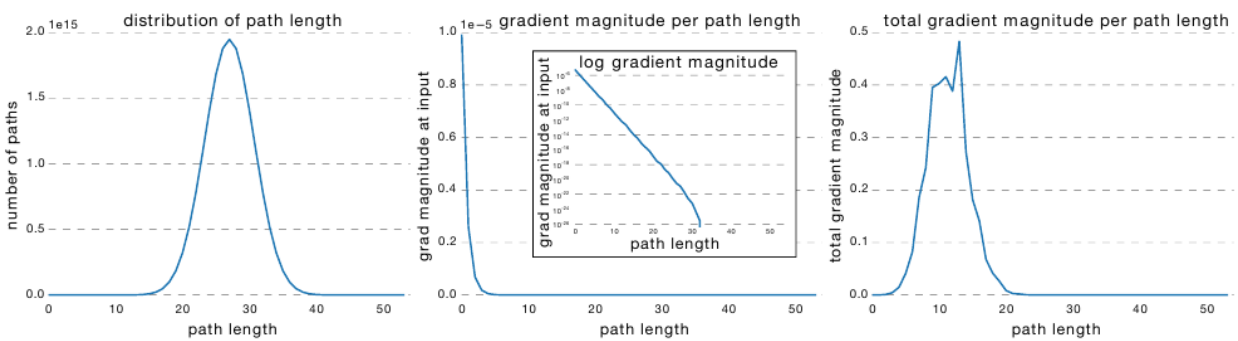
(a) Conventional 3-block residual network (b) Unraveled view of (a)

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.



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`

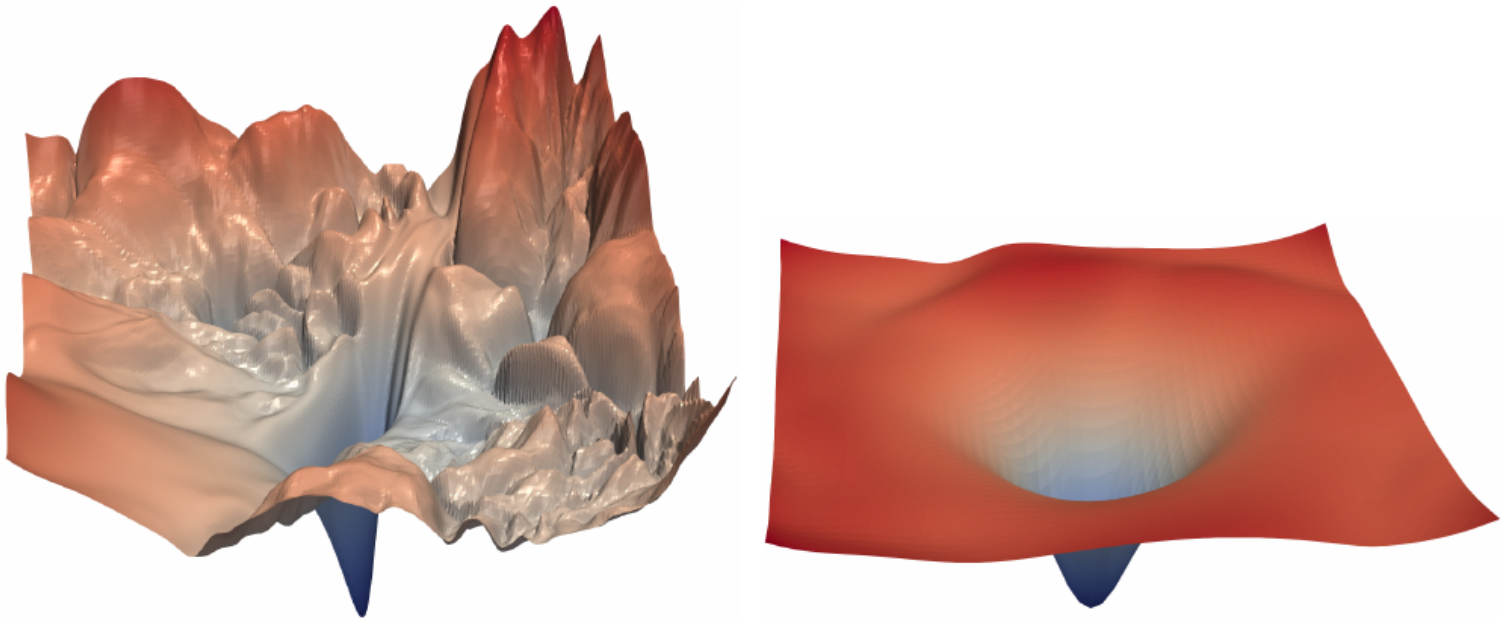
- With their visualization tool, these authors have shown that as a network becomes deeper, its loss function becomes highly chaotic. And that negatively impacts the generalization ability of a network. That is, the network will show a very small training loss and but a significant loss on the test data.

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