2020.04.27

Training Deep Neural Networks

Youngtaek Hong, PhD

Gradient vanishing & exploding

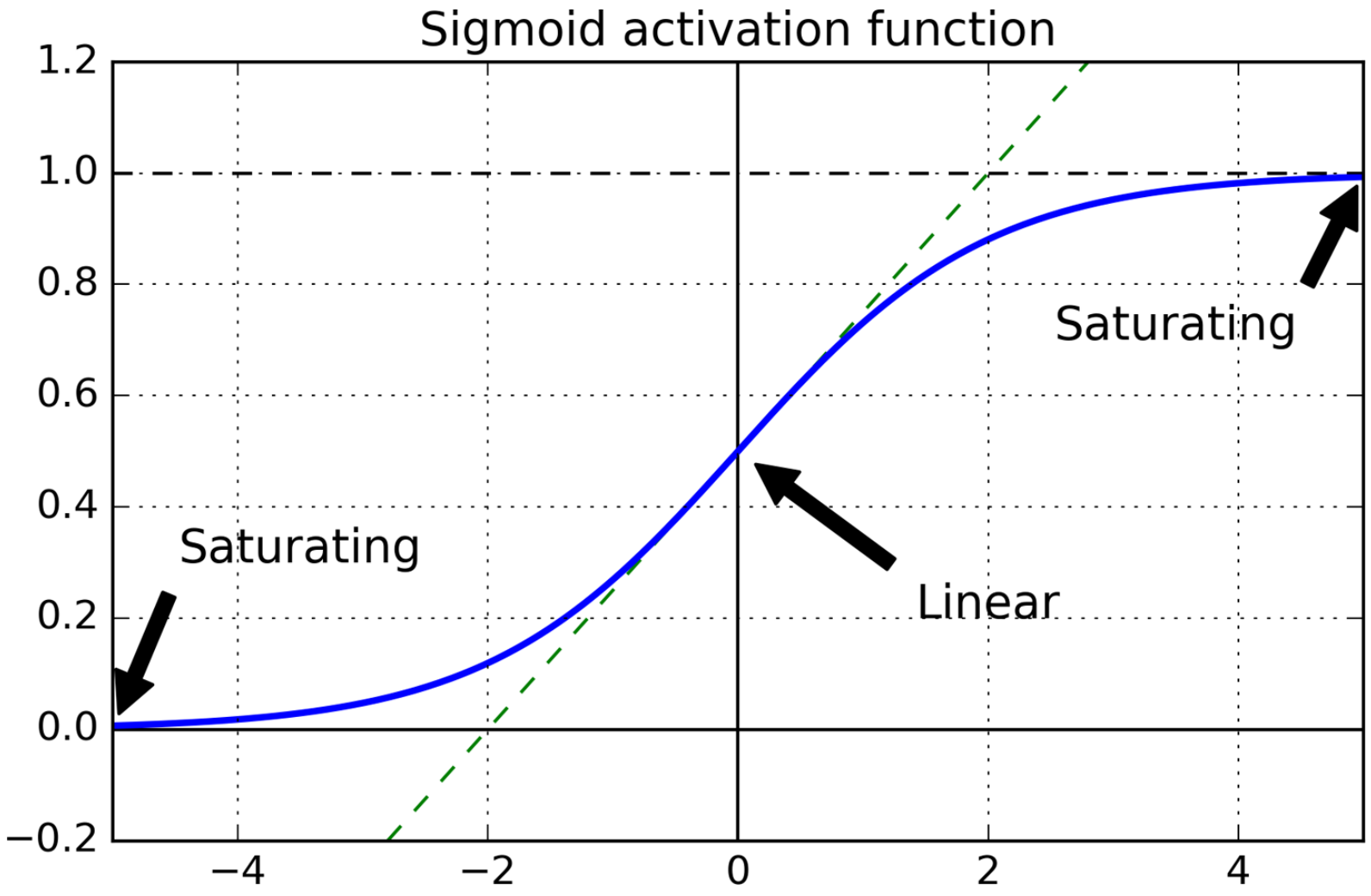
Deep Neural Network 훈련의 어려움

- Vanishing gradient problem, exploding gradient problem.
- Not enough training data for such a large network.
- Training may be extremely slow.
- Overfitting.

The Vanishing/Exploding Gradients Problems

- Backpropagation algorithm 은 출력 계층에서 입력 계층으로 오차를 역전파하여 작동합니다.
- 알고리즘이 네트워크의 각 매개변수에 관한 비용 함수의 기울기를 계산한 후, 이 기울기를 사용하여 각 매개변수를 Gradient Descent Step로 업데이트한다.
- 기울기 값이 레이어를 거칠 때 마다 급격히 감소하는 것을 vanishing gradient
- 기울기 값이 레이어를 거칠 때 마다 급격히 증가하는 것을 exploding gradient

The Vanishing/Exploding Gradients Problems



Xavier Glorot and Yoshua Bengio

- Xavier Glorot and Yoshua Bengio 는 기울기 값이 불안정해 지는 문제를 현저히 완화 시킬 수 있는 방법을 제안하였음.
- The authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs.
- Gradients to have equal variance before and after flowing through a layer in the reverse direction

Xavier initialization

Uniform Xavier initialization: draw each weight, w, from a random uniform distribution in [-x,x] for $x = \sqrt{\frac{6}{inputs + outputs}}$

Normal Xavier initialization: draw each weight, w, from a normal distribution with a mean of 0, and a standard deviation $\sigma = \sqrt{\frac{2}{inputs + outputs}}$

Main idea: Method is not so important. The number of **inputs and outputs** is

365 DataScience

- Keras 는 Glorot initialization with a uniform distribution를 default로 사용함.

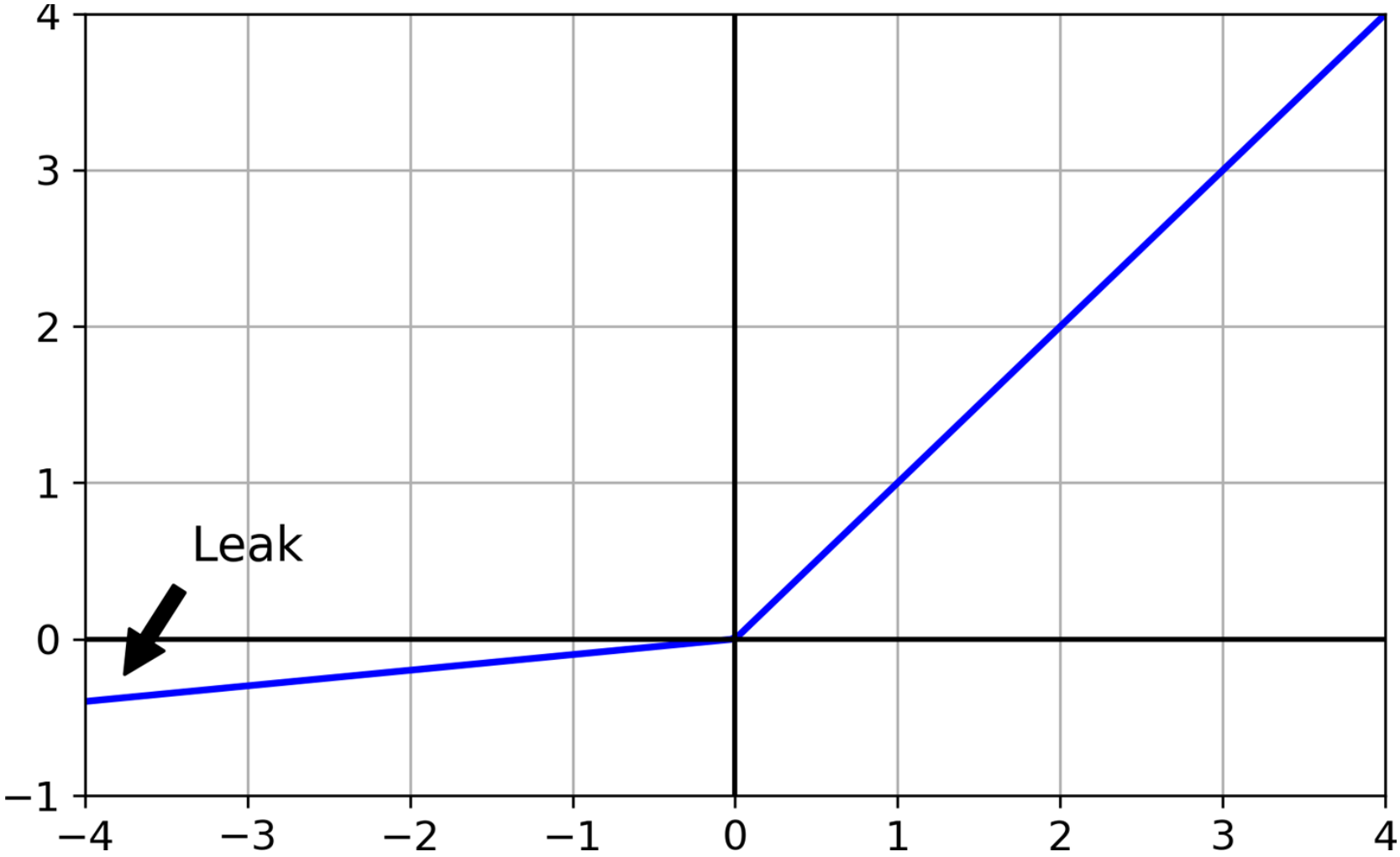
Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Non-saturating Activation Functions

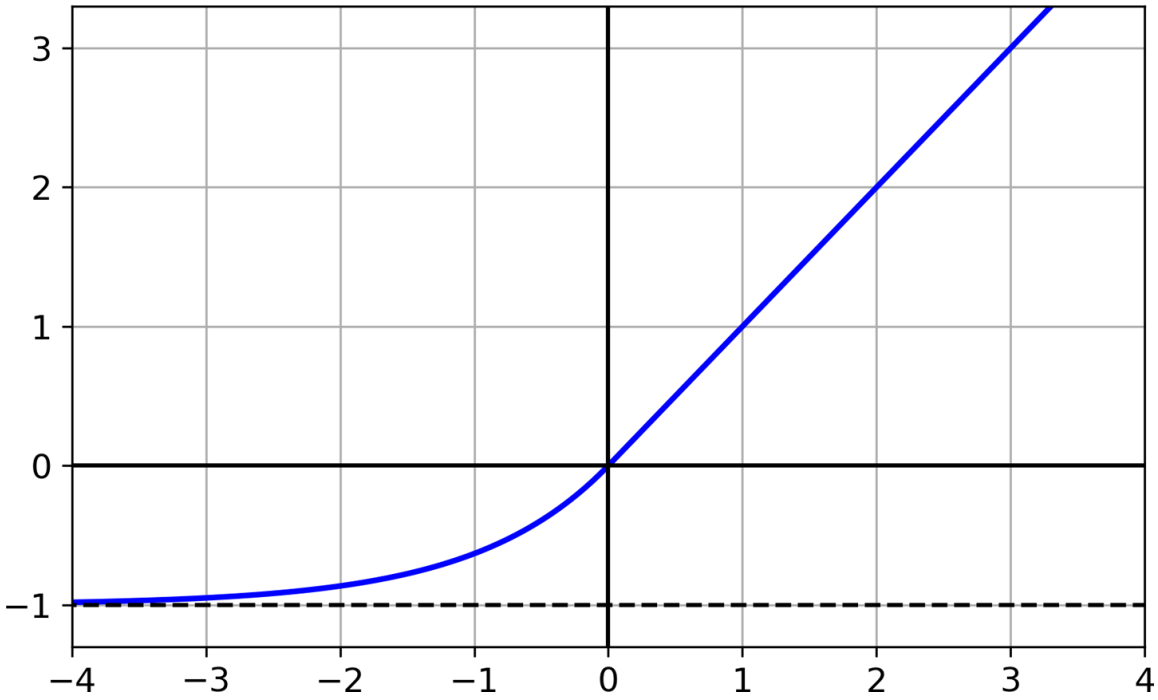
- The ReLU activation function is not perfect.
- It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively "die," meaning they stop outputting anything other than 0 (0 이하의 출력 값이 없는 문제)
- After neuron die → keep outputting zeros
- 이러한 문제를 해결하기 위해 $\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$
- a: 0.3 이 케라스 default
- Parametric leaky ReLU (PReLU) 알파을 training 을 통해 학습
- PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

Leaky ReLU activation function



ELU activation function

- 2015, Djork-Arné Clevert et al proposed a new activation function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants



SELU (Scaled ELU)

- 2017, Günter Klambauer et al. introduced the Scaled ELU (SELU) activation function.
- A neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function, then the network will self-normalize. mean : 0 / std : 1
- Conditions:
    1. The input features must be standardized (mean 0 and standard deviation 1).
    2. kernel_initializer="lecun_normal".
    3. The network’s architecture must be sequential.
    4. some researchers have noted that the SELU activation function can improve performance in convolutional neural nets as well

결론

- in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic.
- runtime latency → leaky ReLU 을 권장
- 0.3 for leaky ReLU 사용을 권장
- ReLU might still be the best choice.

Batch Normalization

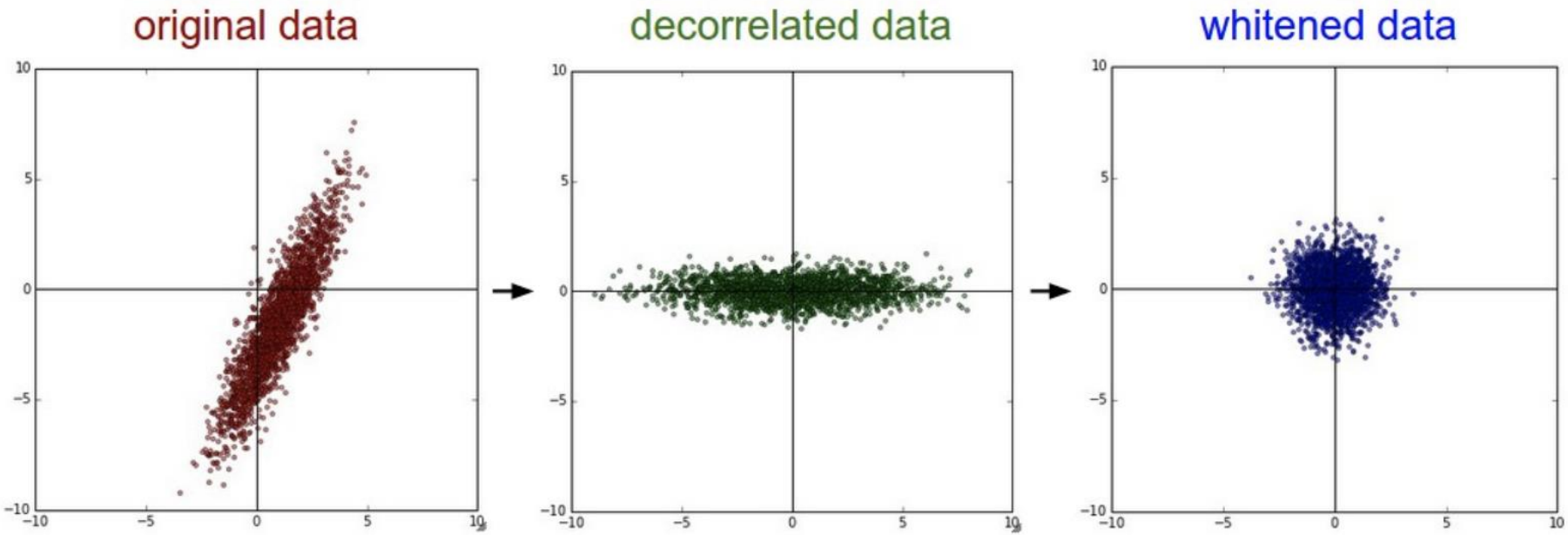
Introduction of Batch normalization

- Vanishing/exploding gradients 문제를 해결하기 위해 He initialization along with ELU 등을 사용해도 vanishing/exploding gradients 문제가 발생하지 않는 다는 보장은 없음
- 이러한 문제를 해결하기 위해, 2015년 Sergey Ioffe and Christian Szegedy 이 *Batch Normalization* (BN) 을 제안하였음.
- This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer.
- zero-centers and normalizes 를 위해 batch 마다 mean과 standard deviation을 계산 → Batch Normalization

Introduction

- Gradient의 불안정화가 일어나는 이유를 'Internal Covariance Shift' 라고 주장하고 있다.
- Internal Covariance Shift라는 현상은 Network의 각 층이나 Activation 마다 input의 distribution이 달라지는 현상을 의미한다.
- 이 현상을 막기 위해서 간단하게 각 층의 input의 distribution을 평균 0, 표준편차 1인 input으로 normalize 시키는 방법을 생각해볼 수 있고, 이는 whitening이라는 방법으로 해결할 수 있다.
- Whitening은 기본적으로 들어오는 input의 feature들을 variance를 1로 만들어주는 작업입니다.

PCA Whitening



PCA / Whitening. **Left**: Original toy, 2-dimensional input data. **Middle**: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix becomes diagonal). **Right**: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.

Introduction

- 문제는 whitening을 하기 위해서는 covariance matrix의 계산과 inverse의 계산이 필요하기 때문에 계산량이 많을 뿐더러, 설상가상으로 whitening을 하면 일부 parameter 들의 영향이 무시된다는 것이다.
- 예를 들어 input u를 받아 x=u+b라는 output을 내놓고 적절한 bias b를 학습하려는 네트워크에서 x에 E[x]를 빼주는 작업을 한다고 생각해보자.
- 그럴 경우 E[x]를 빼는 과정에서 b의 값이 같이 빠지고, 결국 output에서 b의 영향은 없어지고 만다.
- 단순히 E[x]를 빼는 것이 아니라 표준편차로 나눠주는 등의 scaling 과정까지 들어갈 경우 이러한 경향은 더욱 악화될 것이고, 논문에서는 이를 실험적으로 확인했다고 한다.

Introduction

- 와 같은 whitening의 단점을 보완하고, internal covariance shift는 줄이기 위해 Batch Normalization 논문에서는 다음과 같은 접근을 취했다.
- 각각의 feature들이 이미 uncorrelated 되어있다고 가정하고, feature 각각에 대해서만 scalar 형태로 mean과 variance를 구하고 각각 normalize 한다.
- 단순히 mean과 variance를 0, 1로 고정시키는 것은 오히려 Activation function의 nonlinearity를 없앨 수 있다.
- 예를 들어 sigmoid activation의 입력이 평균 0, 분산 1이라면 출력 부분은 곡선보다는 직선 형태에 가까울 것이다. 또한, feature가 uncorrelated 되어있다는 가정에 의해 네트워크가 표현할 수 있는 것이 제한될 수 있다.

Introduction

- training data 전체에 대해 mean과 variance를 구하는 것이 아니라, mini-batch 단위로 접근하여 계산한다.
- 현재 택한 mini-batch 안에서만 mean과 variance를 구해서, 이 값을 이용해서 normalize 한다.

Pseudocode

- 알고리즘의 개요는 위에 서술한 바와 같다. 뉴럴넷을 학습시킬 때 보통 mini-batch 단위로 데이터를 가져와서 학습을 시키는데,
- 각 feature 별로 평균과 표준편차를 구해준 다음 normalize 해주고, scale factor와 shift factor를 이용하여 새로운 값을 만들어준다. 알고리즘의 개요는 다음과 같다.

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

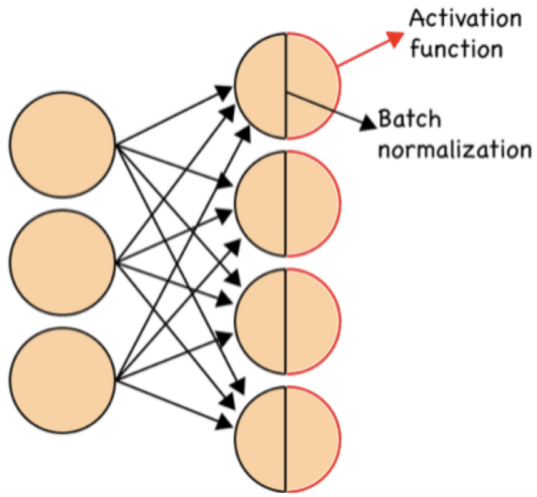
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

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

- 실제로 이 Batch Normalization을 네트워크에 적용시킬 때는, 특정 Hidden Layer에 들어가기 전에 Batch Normalization Layer를 더해주어
- input을 modify해준 뒤 새로운 값을 activation function으로 넣어주는 방식으로 사용한다.



$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

Network with Batch Normalization. 출처: http://mohammadpz.github.io

Batch normalization Inference

- Training Data로 학습을 시킬 때는 현재 보고있는 mini-batch에서 평균과 표준편차를 구하지만, Test Data를 사용하여 Inference를 할 때는 다소 다른 방법을 사용한다.
- mini-batch의 값들을 이용하는 대신 지금까지 본 전체 데이터를 다 사용한다는 느낌으로, training 할 때 현재까지 본 input들의 이동평균 (moving average) 및 unbiased variance estimate의 이동평균을 계산하여 저장해놓은 뒤 이 값으로 normalize를 한다.
- 마지막에 gamma와 beta를 이용하여 scale/shift 해주는 것은 동일하다.

Input: Network N with trainable parameters Θ; subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network

2: **for** $k = 1 \ldots K$ **do**

3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)

4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead

5: **end for**

6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$

7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters

8: **for** $k = 1 \ldots K$ **do**

9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.

10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$

12: **end for**

- 단, 지금까지의 설명은 '일반적인 Network 일 때' 에 한해서 통용된다.
- 만약 Batch Normalization을 CNN에 적용시키고 싶을 경우 지금까지 설명한 방법과는 다소 다른 방법을 이용해야만 한다.
- 먼저, convolution layer에서 보통 activation function에 값을 넣기 전 Wx+b 형태로 weight를 적용시키는데, Batch Normalization을 사용하고 싶을 경우 normalize 할 때 beta 값이 b의 역할을 대체할 수 있기 때문에 b를 없애준다.
- 또한, CNN의 경우 convolution의 성질을 유지시키고 싶기 때문에, 각 channel을 기준으로 각각의 Batch Normalization 변수들을 만든다.

Benefit of Batch Normalization

1. 기존 Deep Network에서는 learning rate를 너무 높게 잡을 경우 gradient가 explode/vanish 하거나, 나쁜 local minima에 빠지는 문제가 있었다.
2. 이는 parameter들의 scale 때문인데, Batch Normalization을 사용할 경우 propagation 할 때 parameter의 scale에 영향을 받지 않게 된다.
3. 따라서, learning rate를 크게 잡을 수 있게 되고 이는 빠른 학습을 가능케 한다.

Benefit of Batch Normalization

1. Batch Normalization의 경우 자체적인 regularization 효과가 있다.
2. 이는 기존에 사용하던 weight regularization term 등을 제외할 수 있게 하며, 나아가 Dropout을 제외할 수 있게 한다
3. (Dropout의 효과와 Batch Normalization의 효과가 같기 때문.) . Dropout의 경우 효과는 좋지만 학습 속도가 다소 느려진다는 단점이 있는데, 이를 제거함으로서 학습 속도도 향상된다.

Experiments

- 같은 learning rate에서 학습을 진행했음에도 불구하고, Batch Normalization을 적용한 네트워크의 에러율이 일반 CNN의 에러율 그래프보다 훨씬 아래에 존재한다.
- 또한, epoch 1만 진행했을 때도 에러율이 3.19% / 1.65%로 반 정도로 차이가 난다.
- 적당히 성능을 비교하기 위해 약 25 epoch 정도만 학습을 진행했는데, 이 과정에서 가장 낮은 Validation Error Rate는 각각 0.92% / 0.78%로 Batch Normalization을 적용한 네트워크 쪽이 더 낮았다.

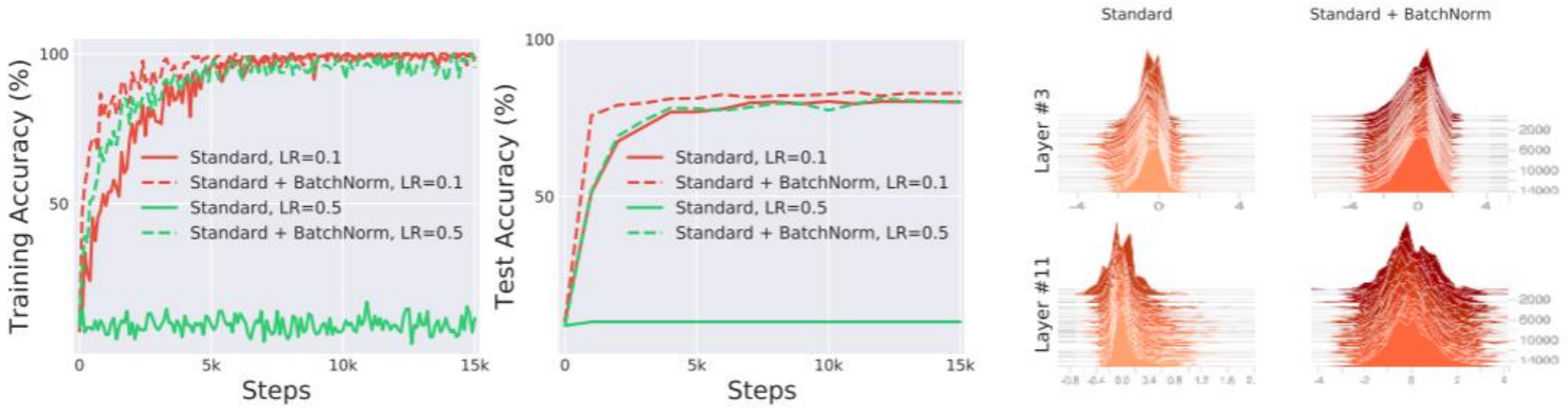


Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A). There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between the performance of the BatchNorm and non-BatchNorm networks is clear, the difference in the evolution of layer input distributions seems to be much less pronounced. (Here, we sampled activations of a given layer and visualized their distribution.)

Example of BN implementation

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Example of BN implementation

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Example of BN implementation

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Example of BN implementation

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

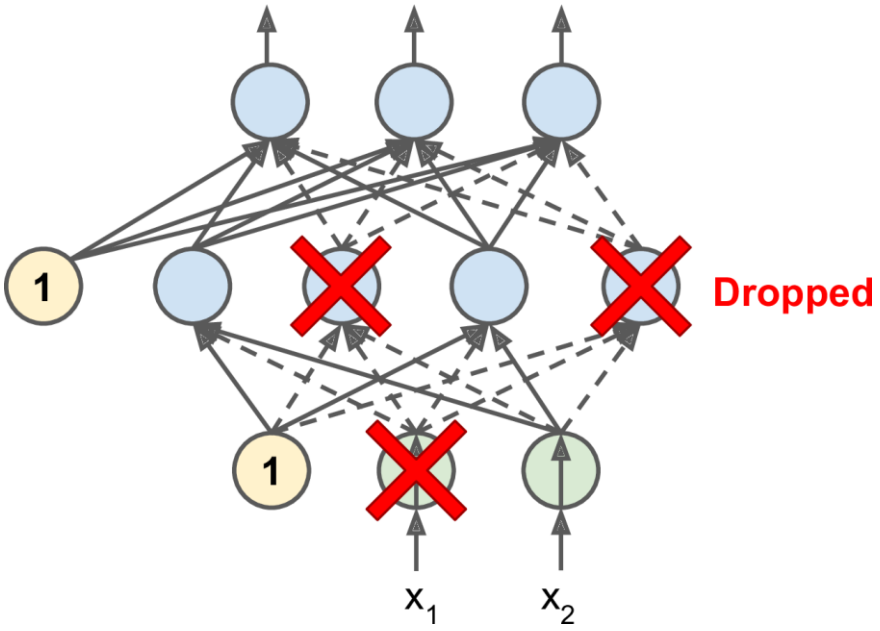
결론

- Batch Normalization에서는 각 layer에 들어가는 input을 normalize 시킴으로써 layer의 학습을 가속하는데, mini-batch의 mean과 variance를 구하여 normalize한다.
- MNIST Dataset으로 실험한 결과 Batch Normalization 기법이 실제로 학습을 가속화하고, 이에 따라 학습의 성능을 향상시킬 수 있다는 것을 실험적으로 확인하였다.

Drop out

Introduce dropout

- *Dropout* is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper[23] by Geoffrey Hinton in 2012 and further detailed in a 2014 paper[24] by Nitish Srivastava et al.,



- In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).
- Since dropout is only active during training, comparing the training loss and the validation loss can be misleading.
- In particular, a model may be overfitting the training set and yet have similar training and validation losses.
- So make sure to evaluate the training loss without dropout (e.g., after training).

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Monte Carlo (MC) Dropout

- In 2016, a paper[25] by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.99]],
      dtype=float32)
```

```
>>> np.round(y_probas[:, :1], 2)
array([[[0.  , 0.  , 0.  , 0.  , 0.  , 0.14, 0.  , 0.17, 0.  , 0.68]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.16, 0.  , 0.2 , 0.  , 0.64]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.01, 0.  , 0.97]],
       [...]
```

```
>>> np.round(y_proba[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.22, 0.  , 0.16, 0.  , 0.62]],
      dtype=float32)
```

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.28, 0.  , 0.21, 0.02, 0.32]],
      dtype=float32)
```