Understanding Batch Normalization

Reference Paper: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift by Ioffe, S. & Szegedy, C.

Motivation

Training deep neural networks with tens of layers is challenging as they can be sensitive to the initial random weights and configuration of the learning algorithm.

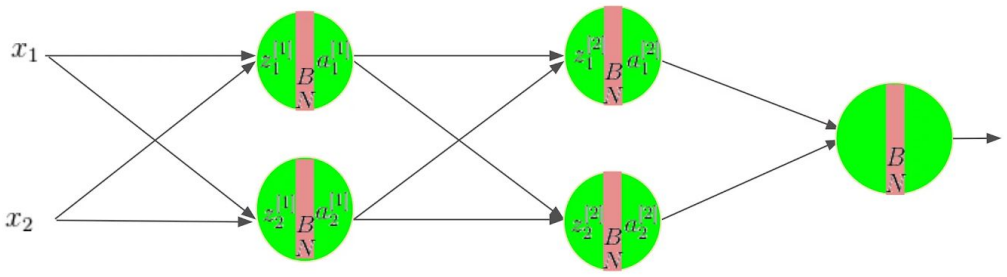
One possible reason for this difficulty is the distribution of the inputs to layers deep in the network may change after each mini-batch when the weights are updated. This can cause the learning algorithm to forever chase a moving target. This change in the distribution of inputs to layers in the network is referred to the technical name “internal covariate shift.” Indeed, the authors found out Internal Covariate Shift exists among hidden layers and it causes slow learning speed and the saturation problem, the resulting vanishing gradients.

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch, which reduce internal covariate shift. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

In addition, batch normalization allows us to use much higher learning rates and be less careful about initialization. It acts as a regularizer, in some cases eliminating the need for Dropout. It also assures input data are in the same range of values.

Process

Batch normalization is done individually at each unit. Figure below shows how it works on a simple network with input features x_1 and x_2 (assuming there is a nonlinear activation layer g before output).



$$z^{[l]} = W^{[l]}a^{[l-1]} \longrightarrow \begin{aligned} \mu^{[l]} &= \frac{1}{m}\sum_i z^{[l](i)} \\ \sigma^{[l]2} &= \frac{1}{m}\sum_i (z^{[l](i)} - \mu^{[l]})^2 \\ z_{norm}^{[l](i)} &= \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\sigma^{[l]2} + \epsilon}} \\ \tilde{z}^{[l](i)} &= \gamma^{[l]} z_{norm}^{[l](i)} + \beta^{[l]} \end{aligned} \longrightarrow a^{[l]} = g^{[l]}(\tilde{z}^{[l]})$$

The superscript (i) corresponds to the i^{th} data in the mini batch, the superscript $[l]$ indicates the l^{th} layer in the network, and the subscript $k$indicates the k^{th} dimension in a given layer in the network. In some places, either of the superscripts or the subscript has been dropped to keep the notations simple.

Batch Normalization is done individually at every hidden unit. Traditionally, the input to a layer $a^{[l-1]}$ goes through an affine transform which is then passed through a non-linearity $g^{[l]}$ such as ReLU or sigmoid to get the final activation a^l from the unit. So,

$a^l = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]})$.

But when Batch Normalization is used with a transform BN, it becomes

$$a^l = g^{[l]}(BN(W^{[l]}a^{[l-1]}))$$

The bias b could now be ignored because its effect is subsumed with the shift parameter β.

The four equations shown in Figure 4 do the following.

1. Calculate mean (μ) of the minibatch.
2. Calculate variance (σ^2) of the minibatch.
3. Calculate z_{norm} by subtracting mean from z and subsequently dividing by standard deviation (σ). A small number, epsilon (ϵ), is added to the denominator to prevent divide by zero.
4. Calculate $\tilde{z}_{norm}$ by multiplying z_{norm} with a scale (γ) and adding a shift (β) and use $\tilde{z}_{norm}$ in place of z as the non-linearity's (e.g. ReLU's) input. The two parameters β and γ are learned during the training process with the weight parameters W.

Python Practice

- The practice is to identify handwriting image on MNIST dataset, using LeNet architecture, combined with customized batch normalization.
- Writting Keras customized layer see https://keras.io/layers/writing-your-own-keras-layers/

Batch Normalization Design

Layers

We apply batch normalization on the input layers before activations, as the authors demonstrated "*The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments we apply it before the nonlinearity since that is where matching the first and second moments is more likely to result in a stable distribution".*

Update ops

It is really important to get the update ops as stated in the Tensorflow documentation because in training time the moving variance and the moving mean of the layer have to be updated. If we don't do this, batch normalization will not work and the network will not train as expected. So we use *tensorflow.python.ops* to introduce moments for the moving average and batch_normalization command (one can find similar structure in the source code of Keras API).

Training phase

It is also useful to declare a placeholder to tell the network if it is in training time or inference time. During training we use the mean and variance of the mini-batch to rescale the input. On the other hand, during inference we use the moving average and variance that was estimated during training. Use keras.backend.learning_phase to declare this status (training = 1, testing = 0).

Other Problems:
https://blog.datumbox.com/the-batch-normalization-layer-of-keras-is-broken/

Model Structure

Input

28x28x1 pixels image

To reduce overfitting, we use another technique known as **Data Augmentation**. Data augmentation rotates, shears, zooms, etc the image so that the model learns to generalize and not remember specific data. If the model overfits, it will perform very well on the images that it already knows but will fail if new images are given to it. Use *keras.preprocessing.image.ImageDataGenerator*

Perform one-hot encoding on the labels (0, 1, …, 9).

Architecture

- **Convolutional #1** outputs 26x26x6

    - **Batch Normalization** outputs 26x26x6
    - **Activation** any activation function, we will use relu
- **Pooling #1** The output shape should be 13x13x6.
- **Convolutional #2** outputs 11x11x16.
    - **Batch Normalization** outputs 11x11x6
    - **Activation** any activation function, we will use relu
- **Pooling #2** outputs 5x5x16.
    - **Flatten** Flatten the output shape of the final pooling layer
- **Fully Connected #1** outputs 120
    - **Batch Normalization** outputs 120
    - **Activation** any activation function, we will use relu
- **Fully Connected #2** outputs 84
    - **Batch Normalization** outputs 84
    - **Activation** any activation function, we will use relu
- **Fully Connected (Logits) #3** outpute 10
    - **Activation** any activation function, we will use softmax

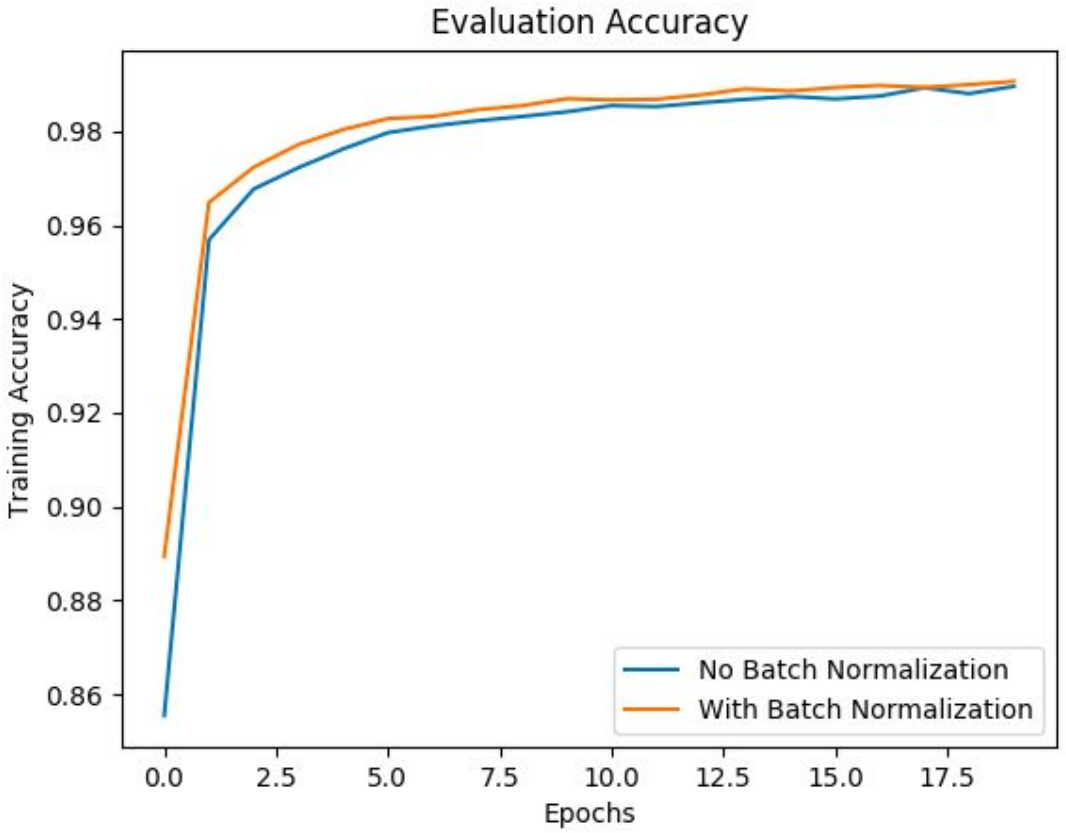
Result

Try Epochs = 20, Batch_Size = 128

training without Batch Normalization took 0:15:43.285176s.
Test loss without Batch Normalization: 0.030737676485381962
Test accuracy without Batch Normalization: 0.9908

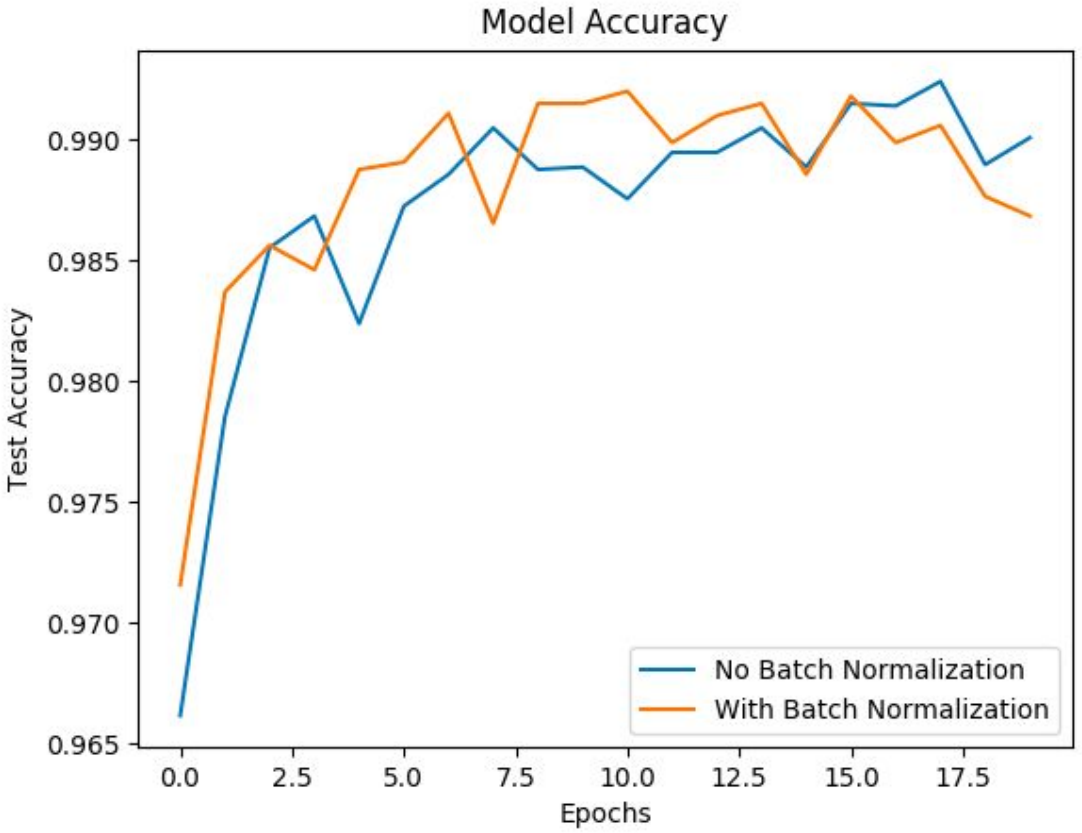
training with Batch Normalization took 0:20:57.456056s.
Test loss with Batch Normalization: 0.03647766597541049
Test accuracy with Batch Normalization: 0.9875

Training Accuracy



Batch Normalization accelerates training, as demonstrated by the authors, "*In a batch-normalized model, we have been able to achieve a training speedup from higher learning rates, with no ill side effects"*.

Testing Accuracy



However, the test performance may have better results with fewer epochs.