Batch Normalisation: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Introduction

“*Training Deep Neural Networks is complicated by the fact that the distribution of each layer’s inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialisation, and makes it notoriously hard to train models with saturating nonlinearities*.”
This is known as internal covariate shift, as described in the paper. The goal of this project is to understand how this problem is mitigated with the help of Batch Normalisation and to reproduce the results as given in the paper with limited resources at hand.

The main issue it causes is the saturation and the vanishing gradients problem, as the W,b changes in the model, there is also a change in the output and hence the input of the next layer. This is best explained in the paper as follows:

“*Consider a layer with a sigmoid activation function $z = g(Wu + b)$ where u is the layer input, the weight matrix W and bias vector b are the layer parameters to be learned, and $g(x) = 1$ $1+\exp(-x)$. As $|x|$ increases, $g'(x)$ tends to zero. This means that for all dimensions of $x = Wu+b$ except those with small absolute values, the gradient flowing down to u will vanish, and the model will train slowly. However, since x is affected by W, b and the parameters of all the layers below, changes to those parameters during training will likely move many dimensions of x into the saturated regime of the nonlinearity and slow down the convergence.*”

Resolving it with ReLU is not enough; we also need to ensure that the initialisation of weights is appropriate, but a more optimal solution would be to ensure that the inputs to not only the neural network but also each of the input layers in a DNN/CNN are normalised. This is achieved via batch normalisation.

Methodology

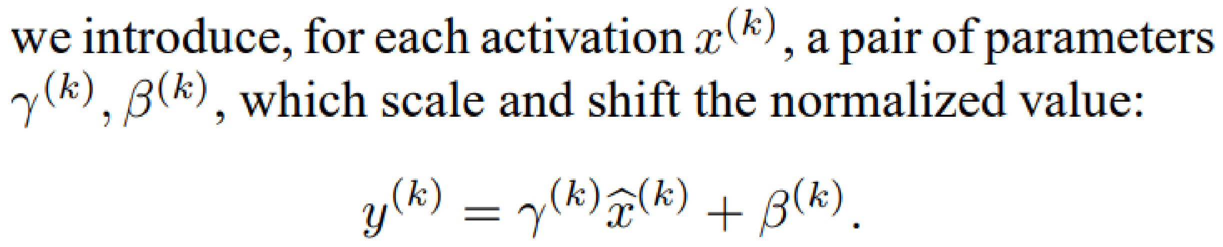
First, understand why naive normalisation doesn’t work while parameter change in SGD.
If we have an input vector x such that $x = u + b$ (b is a bias). Suppose we normalise it by subtracting the mean of the data samples. This would result in $x = x - EX[x] = u + b - EX[u + b]$.
Now for a single gradient step that updates b by Δb. $u + (b + \Delta b) - EX[u + (b + \Delta b)] = u + b + \Delta b - EX[u+b] - \Delta b = u + b - EX[u + b]$.
The above tells us that a small gradient step does not affect the output.

Further, we can consider calculating covariance across the training set. During gradient computation, backprop requires computing derivatives of Cov[x]−1/2. The former is computationally heavy.

Normalisation is applied individually to each network activation, i.e. each neuron and scaled to zero mean and standard deviation 1 with the following formula. The expectation and variance are computed over the mini-batch here.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

But this actually introduces a problem in our training, i.e. the network now won't be able to represent what it originally could, so we apply this before applying non-linearity, and we introduce 2 new parameters which ensure that the model doesn't lose its learning ability.



we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$

Batch normalisation is applied to DNNs and CNN's and the outputs are observed.

For convolutional layers, the mini-batch stats are not applied to each neuron, but they are rather to each feature map, i.e. it has the parameters gamma and beta for C channels.

Below is our own implementation of BN, which is very close to how the PyTorch library itself does BN.

```
Python
# custom batch norm class
class BatchNorm(nn.Module):
    # this will take as input the previous neuron's output and apply
normalisation before giving non-linearity
    def __init__(self,num_features,eps=1e-5,momentum=0.1):
        super().__init__()
        # parameters added for each feature i.e. each previous later neuron
        # setting gamma to be 0 and beta to be 1 i.e. identity transformation
of input
        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta  = nn.Parameter(torch.zeros(num_features))

        # storing running mean and variance to be used for inference time
        self.register_buffer("running_mean", torch.zeros(num_features))
        self.register_buffer("running_var", torch.ones(num_features))

        self.eps = eps
        self.momentum = momentum

    # defining forward layer
    def forward(self, x):
        # If it has batch_size, feature, height and width then it's
convolutional layer
        # Case 2: If it has just batch size and neuron's (features) then fully
connected
        if x.dim() == 4:
            # training for each feature map in CNN
            dims = (0, 2, 3)
            shape = (1, x.size(1), 1, 1)
        else:
            # training for each feature in output
            dims = (0,)
            shape = (1, x.size(1))

        # during training mode compute statistics and in testing mode use train
time running averages
        if self.training:
            # taking mean over feature/feature map
            batch_mean = x.mean(dim=dims)
            batch_var  = x.var(dim=dims, unbiased=False) # use population
variance forumla

            # normalize
            x_hat = (x - batch_mean.view(shape)) /
torch.sqrt(batch_var.view(shape) + self.eps)

            # Update running statistics
            self.running_mean = (
                self.momentum * self.running_mean
                + (1 - self.momentum) * batch_mean.detach() # detach to make
sure no use of gradients for these values
            )
            self.running_var = (
                self.momentum * self.running_var
                + (1 - self.momentum) * batch_var.detach()
            )

        else:
            # during inference using the running averages
            x_hat = (x - self.running_mean.view(shape)) /
torch.sqrt(self.running_var.view(shape) + self.eps)

        # Scale and shift the output to ensure model's learning is not
restricted
        out = self.gamma.view(shape) * x_hat + self.beta.view(shape)
        return out
```

Experiments

Reproducing Section 4.1: Activations over time

As proposed in the paper, to observe the effect of internal covariate shift and how batch normalisation resolves it, we used the MNIST dataset and trained a simple 3-layer deep neural network with 100 neurons, an input image size of 28*28, and 10 output classes.

Architecture Details:

Baseline Model (Without Batch Normalisation)

```
MNIST_NN(
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (bn1): Identity()
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (bn2): Identity()
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (bn3): Identity()
  (fc_out): Linear(in_features=100, out_features=10, bias=True)
)
```

Batch Normalised Model

```
MNIST_NN(
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (bn1): BatchNorm()
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (bn2): BatchNorm()
  (fc3): Linear(in_features=100, out_features=100, bias=True)
  (bn3): BatchNorm()
  (fc_out): Linear(in_features=100, out_features=10, bias=True)
)
```

Non-linear activation function: Sigmoid

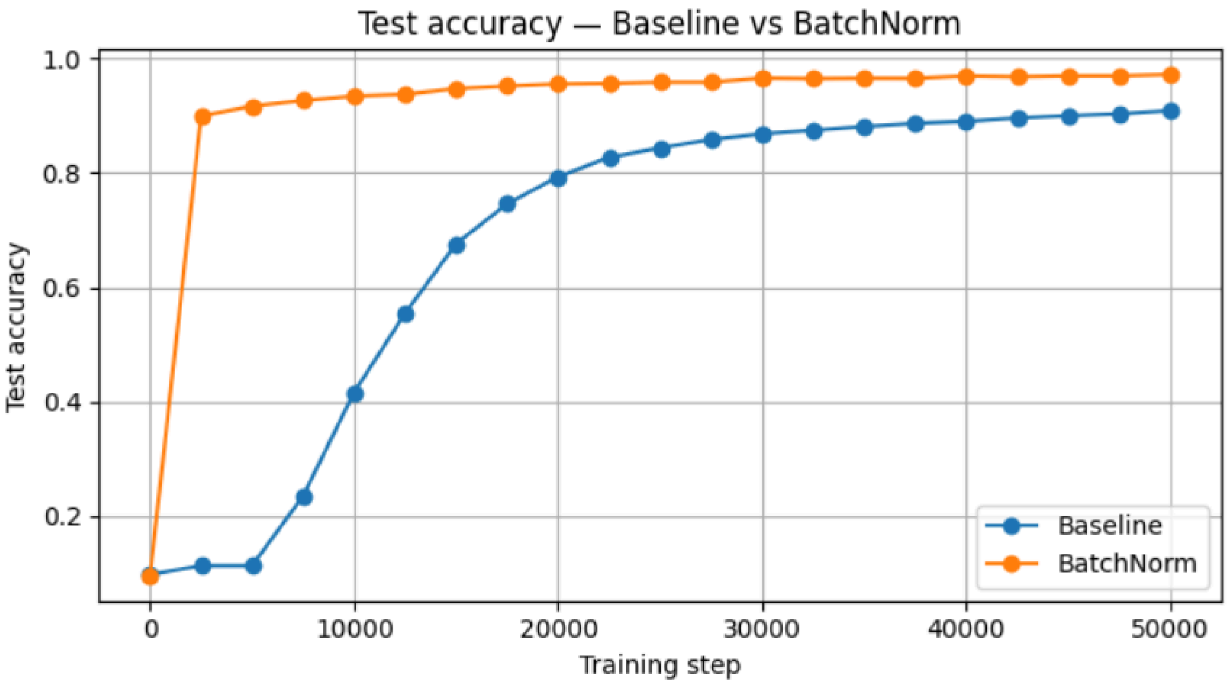
Loss function: Cross-Entropy loss

Number of steps: 50k
Batch size: 1 (this is a mistake!)

Weight initialisation - small random weights

```python
def _init_weights_lecun(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.uniform_(m.weight, -np.sqrt(3/m.in_features),
np.sqrt(3/m.in_features))
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0.0)
            elif isinstance(m, BatchNorm):
                nn.init.constant_(m.running_mean, 0.0)
                nn.init.constant_(m.running_var, 1.0)
```

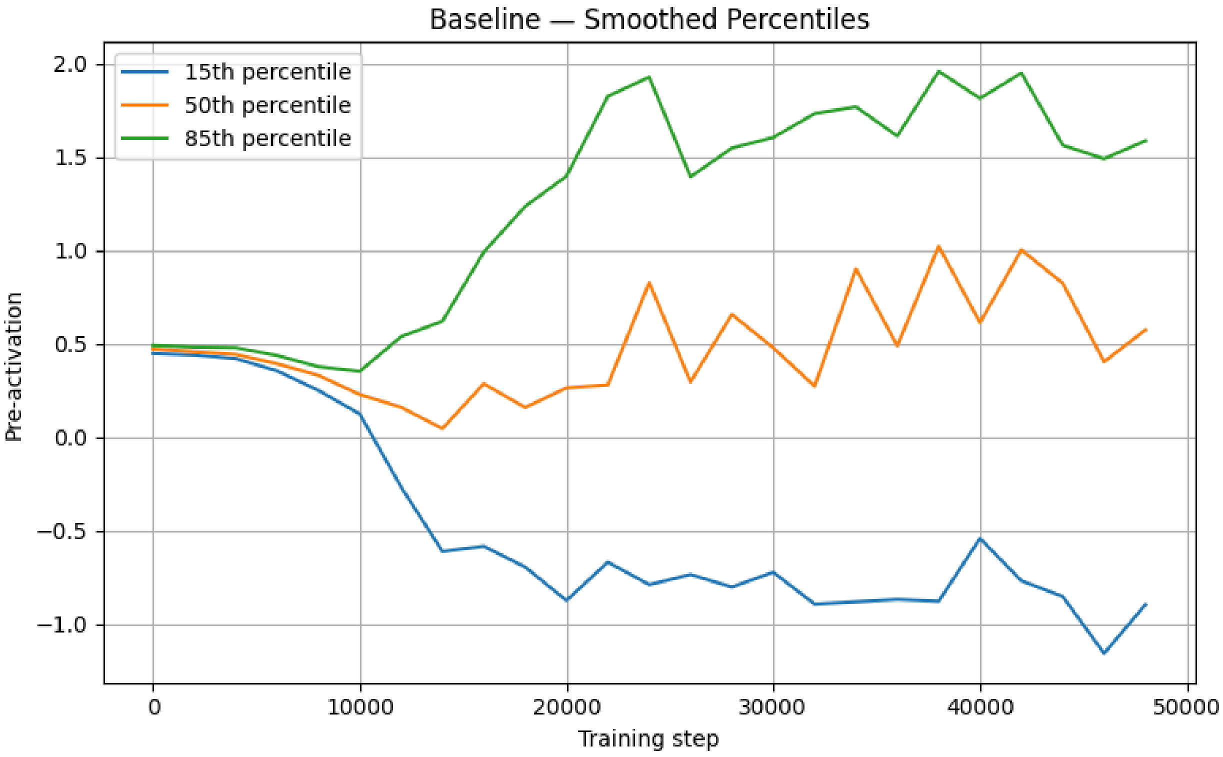
Results:

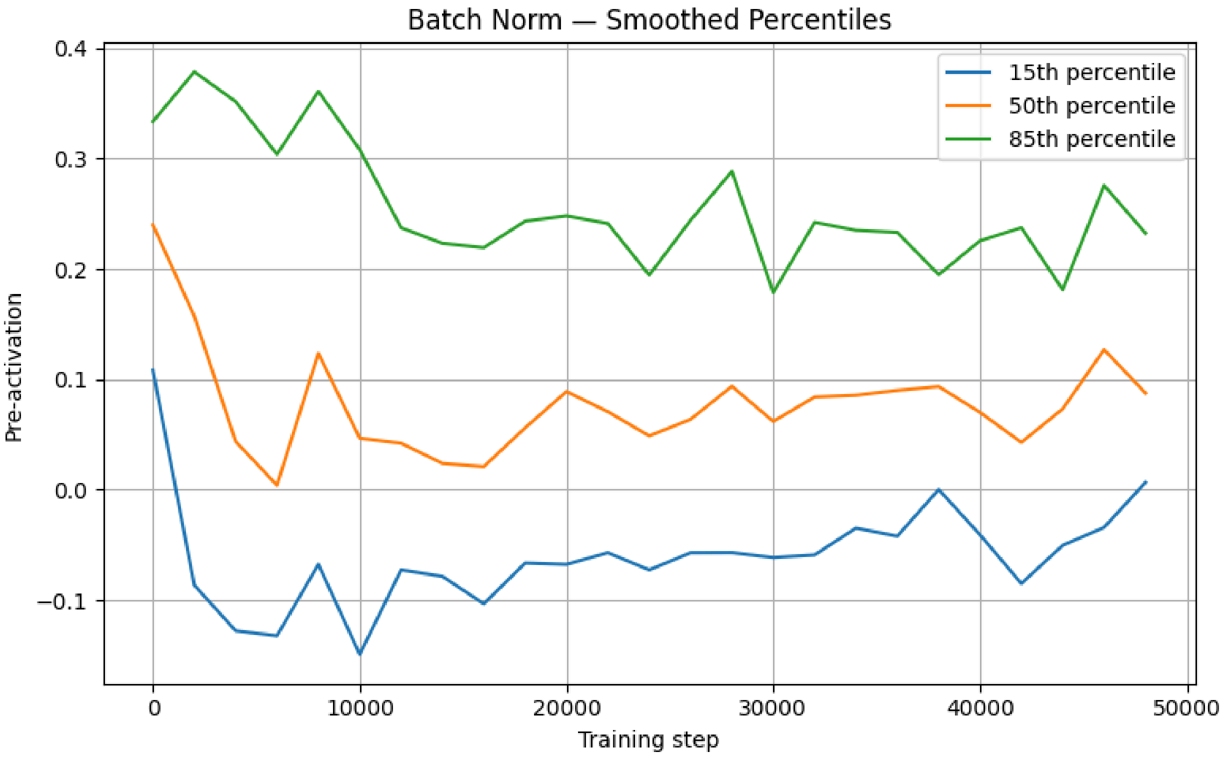


Takeaway:

1. As shown above, we can observe that the batch-normalised model can obtain high accuracy in the 2500th training step itself, whereas in the baseline model, the convergence to the same accuracy is much slower, which is exactly what we hope to observe

   Batch Normalisation helps achieve better accuracy faster.


Baseline — Smoothed Percentiles
15th percentile
50th percentile
85th percentile
Pre-activation
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
10000
20000
30000
40000
50000
Training step


Batch Norm — Smoothed Percentiles
15th percentile
50th percentile
85th percentile
Pre-activation
0.4
0.3
0.2
0.1
0.0
−0.1
0
10000
20000
30000
40000
50000
Training step

Takeaway:

1. As shown above, we observed the difference between the activation of a final layer neuron with and without batch normalisation, and as expected, due to internal covariate shift, there is a variance in the model without batch normalisation vs with batch normalisation. This is the problem that BN helps us mitigate, and hence, we get results faster and better

   Observing the change in input duration for the last layer with and without BN.

Reproducing Section 4.2: ImageNet classification

Dataset used in paper: https://docs.ultralytics.com/datasets/classify/imagenet/

Description:
ImageNet is a large-scale database of annotated images designed for use in visual object recognition research. It contains over 14 million images, with each image annotated using WordNet synsets, making it one of the most extensive resources available for training deep learning models in computer vision tasks.

Please note that, due to its huge size and training being done over 30 million epochs, we were unable to reproduce the same, given the lack of computing power and resources.

Workaround:

Dataset used: CIFAR-10

Description:
The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Architecture used: (mini-inception with only 2 blocks)

```python
class ConvBNAct(nn.Module):
    def __init__(self, in_c, out_c, k, s=1, p=0, use_bn=False,
activation_fn=nn.ReLU):
        super().__init__()
        layers = [nn.Conv2d(in_c, out_c, k, s, p, bias=not use_bn)]
        if use_bn:
            layers.append(CustomBatchNorm2d(out_c))
        layers.append(activation_fn()) # keeping it custom
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

Python

```
class InceptionBlock(nn.Module):
    def __init__(self, in_c, use_bn=False, activation_fn=nn.ReLU):
        super().__init__()

        # shrink all widths
        self.b1 = ConvBNAct(in_c, 8, 1, use_bn=use_bn,
activation_fn=activation_fn)

        self.b2 = nn.Sequential(
            ConvBNAct(in_c, 8, 1, use_bn=use_bn, activation_fn=activation_fn),
            ConvBNAct(8, 16, 3, p=1, use_bn=use_bn,
activation_fn=activation_fn)
        )

        self.b3 = nn.Sequential(
            ConvBNAct(in_c, 8, 1, use_bn=use_bn, activation_fn=activation_fn),
            ConvBNAct(8, 16, 3, p=1, use_bn=use_bn,
activation_fn=activation_fn),
            ConvBNAct(16, 16, 3, p=1, use_bn=use_bn,
activation_fn=activation_fn)
        )

        self.b4 = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            ConvBNAct(in_c, 8, 1, use_bn=use_bn, activation_fn=activation_fn)
        )

    def forward(self,x):
        return torch.cat([self.b1(x), self.b2(x), self.b3(x), self.b4(x)],
dim=1)
```

Python

```python
class TinyInception(nn.Module):
    def __init__(self, num_classes=10, use_bn=False, activation_fn=nn.ReLU):
        super().__init__()

        self.stem = ConvBNAct(3, 16, 3, p=1, use_bn=use_bn,
activation_fn=activation_fn)
        self.inc1 = InceptionBlock(16, use_bn=use_bn,
activation_fn=activation_fn)
        self.inc2 = InceptionBlock(48, use_bn=use_bn,
activation_fn=activation_fn)
        self.pool = nn.AdaptiveAvgPool2d(1)

        # Use dropout only when BN is OFF
        self.drop = nn.Dropout(0.4) if not use_bn else nn.Identity()

        self.fc = nn.Linear(48, num_classes)

    def forward(self, x):
        x = self.stem(x)
        x = self.inc1(x)
        x = self.inc2(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)
```

What does the Inception block do?

1. It applies multiple filters of different sizes (1×1, 3×3, 5×5) to the same input at the same time.
2. Each filter size captures different types of features - small filters catch fine details, bigger filters capture broader patterns.
3. A 1×1 convolution reduces channel count, so the block stays efficient.
4. Finally, it concatenates all feature maps, giving the network a rich mix of features at multiple scales.

Final layer: Uses softmax and cross-entropy loss function.

Please note: The application of batch normalisation is done at the channel layer in CNN as described in the paper, and not for each activation, and this is applied before the non-linearity.

Section 4.2.2 Single-Network Classification

Experiment 1:

Training condition:

```
Batch Normalisation: False
Dropout: True - 40%
Number of epochs: 100
Learning rate: 0.0015
```

Results:

```
Validation Accuracy: 58%
```

Takeaway:

> Why do we need dropouts? Without BN, we need some way to regularise the network to prevent overfitting and hence dropout is required, and as we will see in graph, the training is much slower in comparison to BN models

Experiment 2:

Training condition:

```
Batch Normalisation: True
Dropout: False
Number of epochs: 30
Learning rate: 0.0015
```

Results:

```
Validation Accuracy: 68%
Same as Baseline model accuracy achieved at: Epoch 11
```

Takeaway:

> With batch normalisation, the training is much faster as we don’t run into the vanishing gradients problem, and it acts as a regularizer on itself.

Experiment 3:

Training condition:

```
Batch Normalisation: True
Dropout: False
Number of epochs: 30
Learning rate: 0.0075
```

Results:

```
Validation Accuracy: 68%
Same as Baseline model accuracy achieved at: Epoch 6
```

Takeaway:

With batch normalisation, we always scale the input to the next model by normalising it. This enables less impact of the weights, i.e. no matter how big the weight is, using u = wx+b, we will normalise it for the entire batch, and the next layer will always see scaled results, so we can increase the learning rate and still obtain stable results.

Experiment 4:

Training condition:

```
Batch Normalisation: True
Dropout: False
Number of epochs: 30
Learning rate: 0.045
Regularisation: True
```

Results:

```
Validation Accuracy: 73%
Same as Baseline model accuracy achieved at: Epoch 3
```

Takeaway:

Large learning rate → Weight values increase, let’s combat it with weight decay → L2 regularisation. The model trains much faster.

Experiment 5:

Training condition:

```
Batch Normalisation: True
Dropout: False
```

```
Number of epochs: 30
Learning rate: 0.0075
Non-linearity: Sigmoid
```

Results:

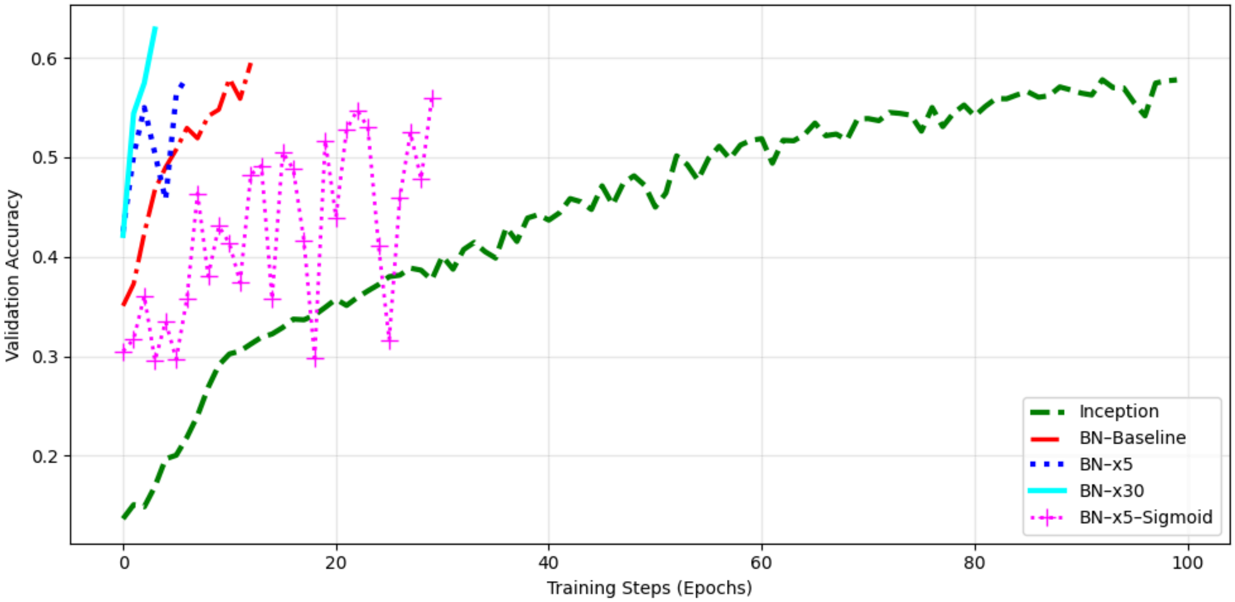
```
Validation Accuracy: 56%
Same as Baseline model accuracy achieved at: Couldn't reach
```

Takeaway:

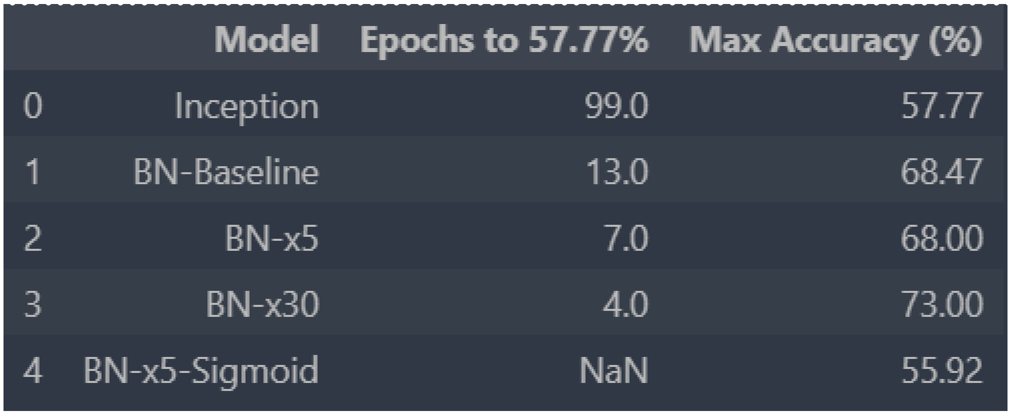
Without batch normalisation and sigmoid as non-linearity, the model is unable to learn at all due to the vanishing gradient problem, but with BN, it can learn, slowly but doesn't stop learning at least.

Training is better with RELU, but with BN, it at least doesn't stop training completely.

Graph:



Result:



	Model	Epochs to 57.77%	Max Accuracy (%)
0	Inception	99.0	57.77
1	BN-Baseline	13.0	68.47
2	BN-x5	7.0	68.00
3	BN-x30	4.0	73.00
4	BN-x5-Sigmoid	NaN	55.92

Section 4.2.3 Ensemble Classification

Paper: Reports the best output for the top-5 validation error on the ImageNet Large Scale competition using a combination of 6 different ensemble models, each trained independently as per the research paper: `Going deeper with convolutions`.

We can't do the same, but based on the description in the paper, we defined 6 different models. The main thing which the referred paper did was shuffle the dataset before training individual models and applying transformations on the input. They did multi-crop analysis on the 256*256 pixel image with 8-100% rescaling, but the same can't be applied as our image size is 32*32 pixels, so we did random crop and applied minimal photometric distortions, as BN already provides a regularisation effect. The function below defines the loading of train and test data.

```python
import torchvision.transforms as T

# performing random corp and photometric distortion on training data
def cifar10_inception_style_transform():
    transform = T.Compose([
        # randomCrop
        T.RandomCrop(32, padding=4),

        # flipping image
        T.RandomHorizontalFlip(),
        # photometric distortions- but small as suggestion
        T.ColorJitter(
            brightness=0.2,
            contrast=0.2,
            saturation=0.2,
            hue=0.05
        ),

        T.ToTensor(),

        # Normalize
        T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    return transform
```

Here are the 6 different models we decided to train.
Please note that the paper didn't provide details on how each model had its parameters set, so we made the following assumptions.

Python

```
def build_ensemble_models():
    models = []

    configs = [
        # dropout 5% - 10%
        {"dropout_rate": 0.05, "increase_init": False, "final_bn": False},
        {"dropout_rate": 0.10, "increase_init": False, "final_bn": False},

        # increased weight init
        {"dropout_rate": 0.05, "increase_init": True, "final_bn": False},
        {"dropout_rate": 0.10, "increase_init": True, "final_bn": False},

        # using batchNorm in the final linear layer
        {"dropout_rate": 0.05, "increase_init": False, "final_bn": True},
        {"dropout_rate": 0.10, "increase_init": False, "final_bn": True},
    ]

    for cfg in configs:
        model = TinyInception(
            num_classes=10,
            use_bn=True,
            dropout_rate=cfg["dropout_rate"],
            increase_init=cfg["increase_init"],
            final_bn=cfg["final_bn"]
        )
        models.append({"model":model,"config":cfg})

    return models
```

Results: `Ensemble Validation Accuracy = 73.64%`

Takeaway

> This is better than the accuracy of individual models, but it still doesn't give us a significant improvement. The reason of mentioning in the paper was their ability to achieve state-of-the-art performance in terms of the ImageNet competition, but for us, as we are using a different dataset and model, the same can't be deducted, but we learnt how to implement and train multiple models independently and observed about 2-3% improvement in accuracy than individual models.

Failed Experiments

Failure 1: TinyImageNet + Inception

Initially, we tried to train our tiny Inception model on the Tiny Imagenet dataset but didn't succeed because of the huge training set.

TinyImageNet Description:
Tiny ImageNet contains 100000 images of 200 classes (500 for each class), downsized to 64×64 colored images. Each class has 500 training images, 50 validation images, and 50 test images.

Initial training mode: Kaggle → Memory crash, notebook shutdown issues.
Moved to Colab.

In Colab, we optimised the model by using mixed precision and the minimum possible batch size, but still failed to completely train a single model. Each time we tried to optimise everything, including scaling the input image, i.e. normalising it, we failed and hence had to choose another dataset for performing the results.

More training details with memory-optimised code are available on GitHub.

Failure 2: CIFAR-10 + LeNet-5

As available everywhere online, to train the CIFAR-10 dataset, we used the LeNet-5 architecture, but while measuring the training and validation accuracy after each epoch, we noticed that the training accuracy was improving, but the validation accuracy was almost stagnant, which implied the model was overfitting. We carefully checked our code and compared it with online resources, but couldn’t see any problem in the training loop, so we moved on to a combination of CIFAR-10 + TinyInception architecture.

Need to understand this failure better. Why is this the case?

Training and validation accuracy for the trained model, more code available on GitHub

```
None
Epoch [1/20] - Train Acc: 42.43% - Val Acc: 49.64%
Epoch [2/20] - Train Acc: 52.95% - Val Acc: 54.48%
Epoch [3/20] - Train Acc: 57.29% - Val Acc: 57.25%
Epoch [4/20] - Train Acc: 60.65% - Val Acc: 58.08%
Epoch [5/20] - Train Acc: 62.87% - Val Acc: 58.31%
Epoch [6/20] - Train Acc: 65.00% - Val Acc: 59.70%
Epoch [7/20] - Train Acc: 66.68% - Val Acc: 59.45%
Epoch [8/20] - Train Acc: 68.00% - Val Acc: 59.42%
Epoch [9/20] - Train Acc: 69.18% - Val Acc: 61.63%
Epoch [10/20] - Train Acc: 70.14% - Val Acc: 60.90%
Epoch [11/20] - Train Acc: 71.18% - Val Acc: 60.20%
Epoch [12/20] - Train Acc: 72.07% - Val Acc: 60.68%
Epoch [13/20] - Train Acc: 72.86% - Val Acc: 60.71%
Epoch [14/20] - Train Acc: 73.54% - Val Acc: 60.96%
Epoch [15/20] - Train Acc: 74.50% - Val Acc: 60.51%
Epoch [16/20] - Train Acc: 74.73% - Val Acc: 60.91%
Epoch [17/20] - Train Acc: 75.70% - Val Acc: 59.62%
Epoch [18/20] - Train Acc: 76.16% - Val Acc: 60.33%
Epoch [19/20] - Train Acc: 77.08% - Val Acc: 59.78%
Epoch [20/20] - Train Acc: 77.27% - Val Acc: 59.39%
Epoch [21/20] - Train Acc: 77.71% - Val Acc: 59.55%
Epoch [22/20] - Train Acc: 78.38% - Val Acc: 59.62%
Epoch [23/20] - Train Acc: 78.55% - Val Acc: 58.44%
Epoch [24/20] - Train Acc: 79.06% - Val Acc: 59.53%
Epoch [25/20] - Train Acc: 79.65% - Val Acc: 59.05%
Epoch [26/20] - Train Acc: 79.92% - Val Acc: 58.90%
Epoch [27/20] - Train Acc: 80.26% - Val Acc: 59.48%
Epoch [28/20] - Train Acc: 80.49% - Val Acc: 57.83%
Epoch [29/20] - Train Acc: 80.72% - Val Acc: 58.78%
Epoch [30/20] - Train Acc: 81.11% - Val Acc: 58.44%
```

Conclusion

Without Batch Normalisation:
Problems

1. Internal covariate shift: The inputs to each layer change drastically as the weights are updated, so the model keeps learning again and again on continuously changing data.
2. Huge problems with vanishing and exploding gradients, and the inability to train with sigmoid and those types of activation functions.
3. Slower and sometimes even no training happens due to small weight updates.
4. High dependency on the initial weights of the network.
5. Can't train faster with higher learning rates.

With Batch Normalisation:
Solutions:

1. Solves internal covariate shift by not only normalising the input layer, which we usually do, but also the input to each network, i.e. all the inputs/neurons feeding into the network, their values during forward propagation are normalised.
2. Doesn't limit the learning capability of the model: We apply an affine transformation on the output of batch normalised value for the model to capture any relationship like before, and this is applied before non-linearity, i.e. to ensure that the model is able to learn non-linear functions as well.
3. It adds 2 parameters to the training, which are learned, i.e. output = yx+c, where y and c are the parameters being learned by the network, but despite an increase in the number of parameters, the model still learns faster than the baseline model.

4. As the normalisation is applied on mini-batches, it acts as a regularizer, i.e. reducing the need for dropout or other external ways of normalising the inputs.

Team Details

Juhi Bhojani	25M0764
Pratik Tadvi	25M0839
Mithil Vasava	25M0840
Revathy P	25M0768
Harshay Bairagi	25M0792

GitHub Link: https://github.com/Juhibhojani/Batch-Normalization