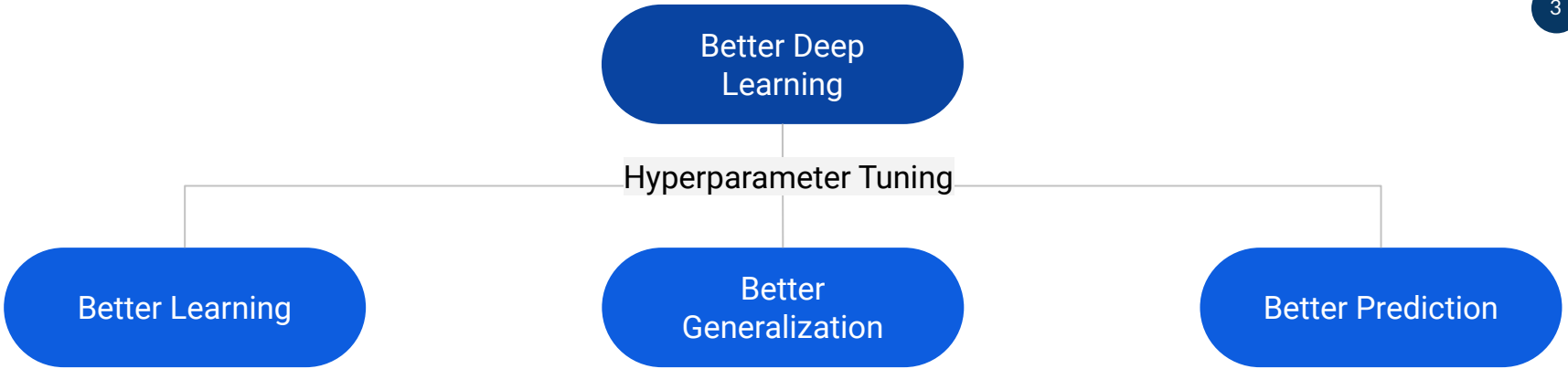
Better Deep Learning

Hyperparameter Tuning & Batch Normalization

Better Learning+Better Generalization



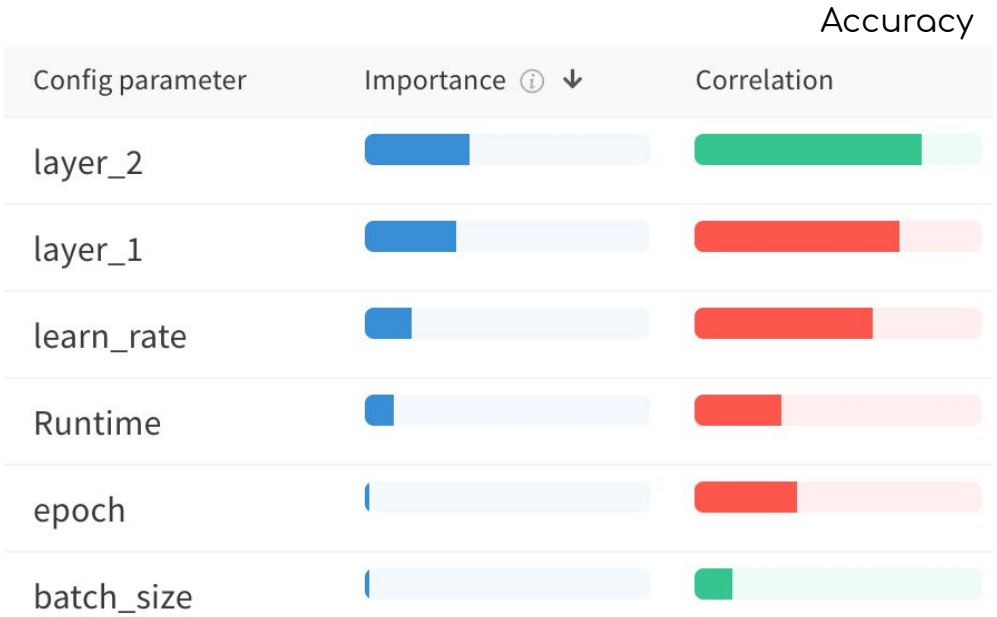
1. **Configure capacity with node and layers**
2. **Configure gradient precision with batch size**
3. **Configure what to optimize with loss functions**
4. **Configure speed of learning with learning rate**
5. Stabilize learning with data scaling
6. Fix vanishing gradient with relu
7. Fix exploding gradient with gradient clipping
8. Accelerate learning with batch normalization

1. Fix overfitting with regularization
2. Penalize large weights with weight regularization
3. Force small weights with weight constraint
4. Decouple layers with dropout
5. Promote robustness with noise
6. Halt training at the right time with early stopping

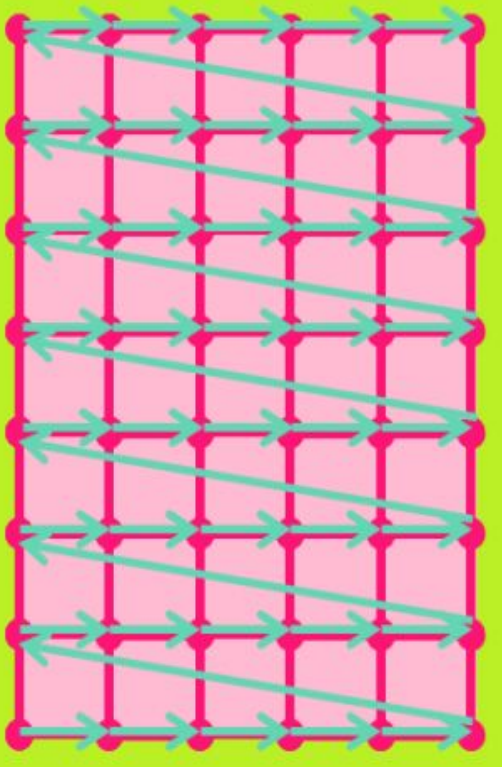
Hyperparameter Tuning

Hyperparameters

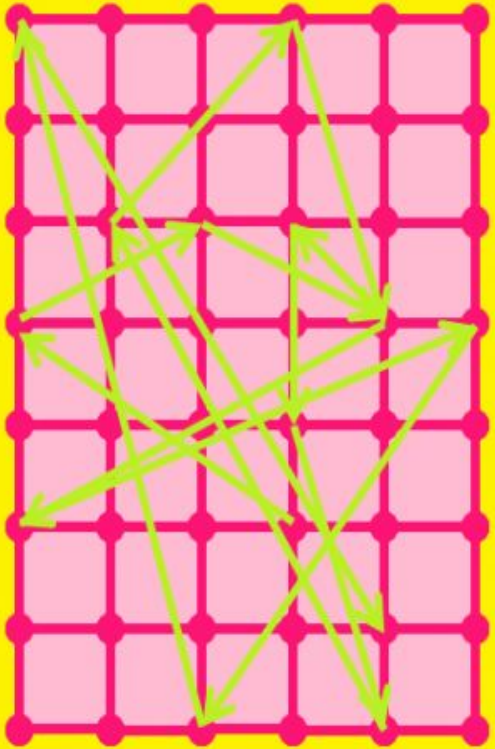
η
β
#hidden_units
#batch_size
$\beta_1, \beta_2, \epsilon$
#layers
Learning_rate_decay
L2, Dropout
Gradient_Clipping
Noise
...



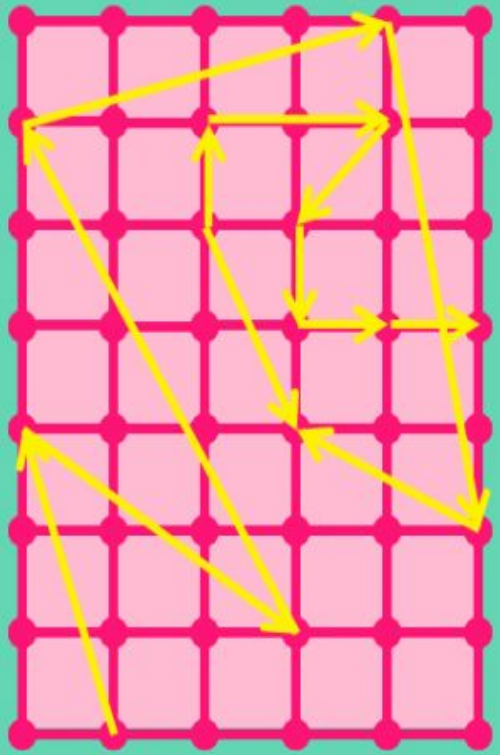
GRID



RANDOM

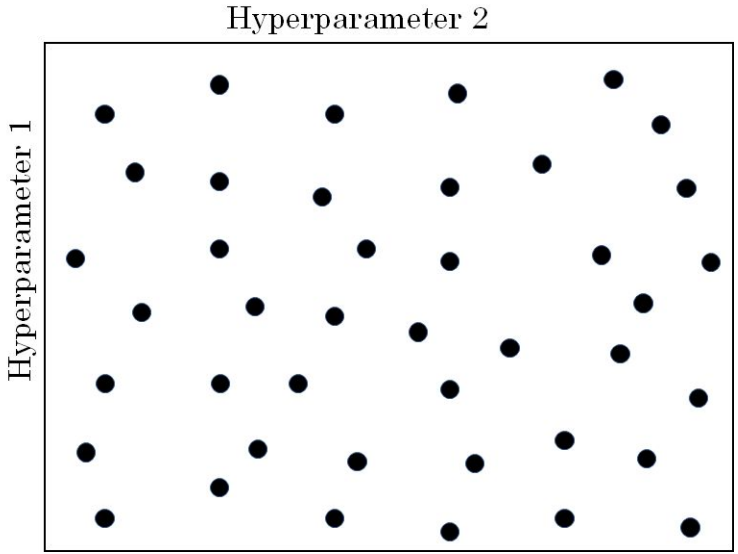


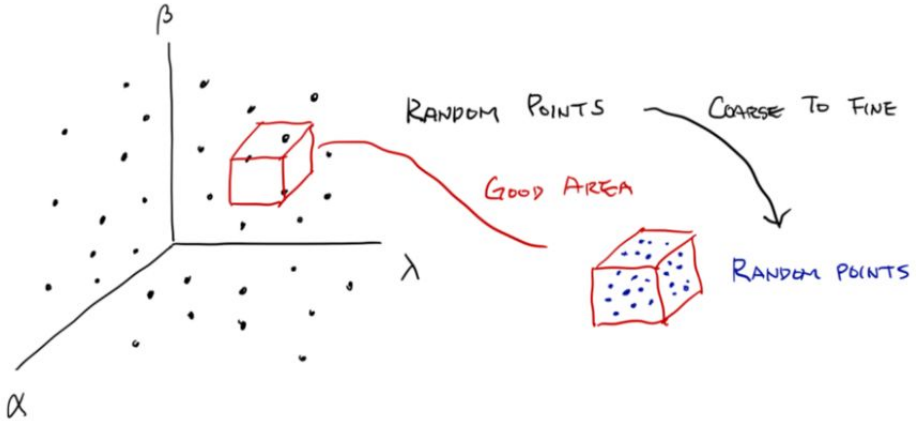
BAYESIAN



https://quantdare.com

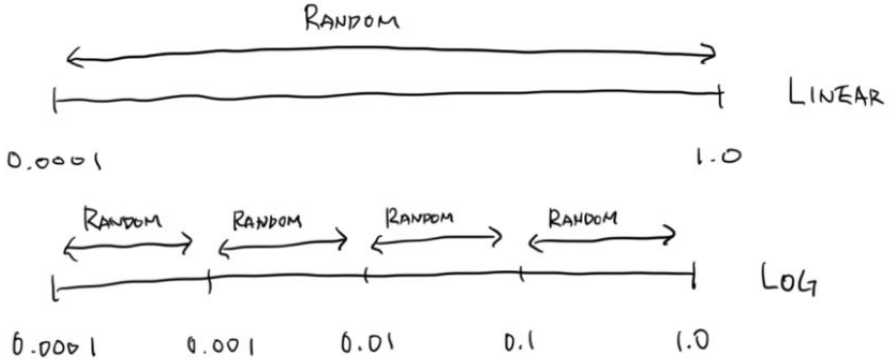
Try random values: don't use a grid





Feurer et al. Auto-Sklearn 2.0: The Next Generation, Jul. 2020

Picking Hyperparameters at Random



$n^{[l]}$ → np.random.randint(50,100)
L → np.random.randint(2,5)

$$r = \log_{10} 1 \rightarrow 10^r = 10^0 \rightarrow r = 0$$

$$r = \log_{10} 0.0001 \rightarrow 10^r = 10^{-4} \rightarrow r = -4$$

r = -4 np.random.rand( )
η = np.power(10,r)

Hyperparameters for Exponentially Weighted Averages

					$r \in [-3,-1]$
ß	0.9	0.9005	0.999	0.9995	$1 - 10^r$
1-ß	0.1	0.0995	0.001	0.0005	10^r
Average 1/(1-ß)	10	10.05	1000	2000	

ß

random random

0.9 0.99 0.999

1-ß

random random

0.1 0.01 0.001

Keras Tuner

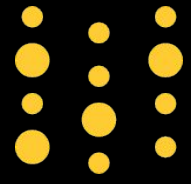
Weights and Biases

```
from tensorflow import keras
from tensorflow.keras import layers
from kerastuner.tuners import RandomSearch


def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Dense(units=hp.Int('units',
                                        min_value=32,
                                        max_value=512,
                                        step=32),
                           activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))
    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Choice('learning_rate',
                      values=[1e-2, 1e-3, 1e-4],
                      sampling='LOG')),
        loss='binary_crossentropy',
        metrics=['accuracy'])
    return model
```

https://keras-team.github.io/keras-tuner/

```
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=3,
    directory='my_dir',
    project_name='helloworld')
```

```
tuner.search(train_x,
             train_y,
             epochs = 500,
             verbose=0,
             batch_size=32,
             validation_data = (test_x, test_y))
```

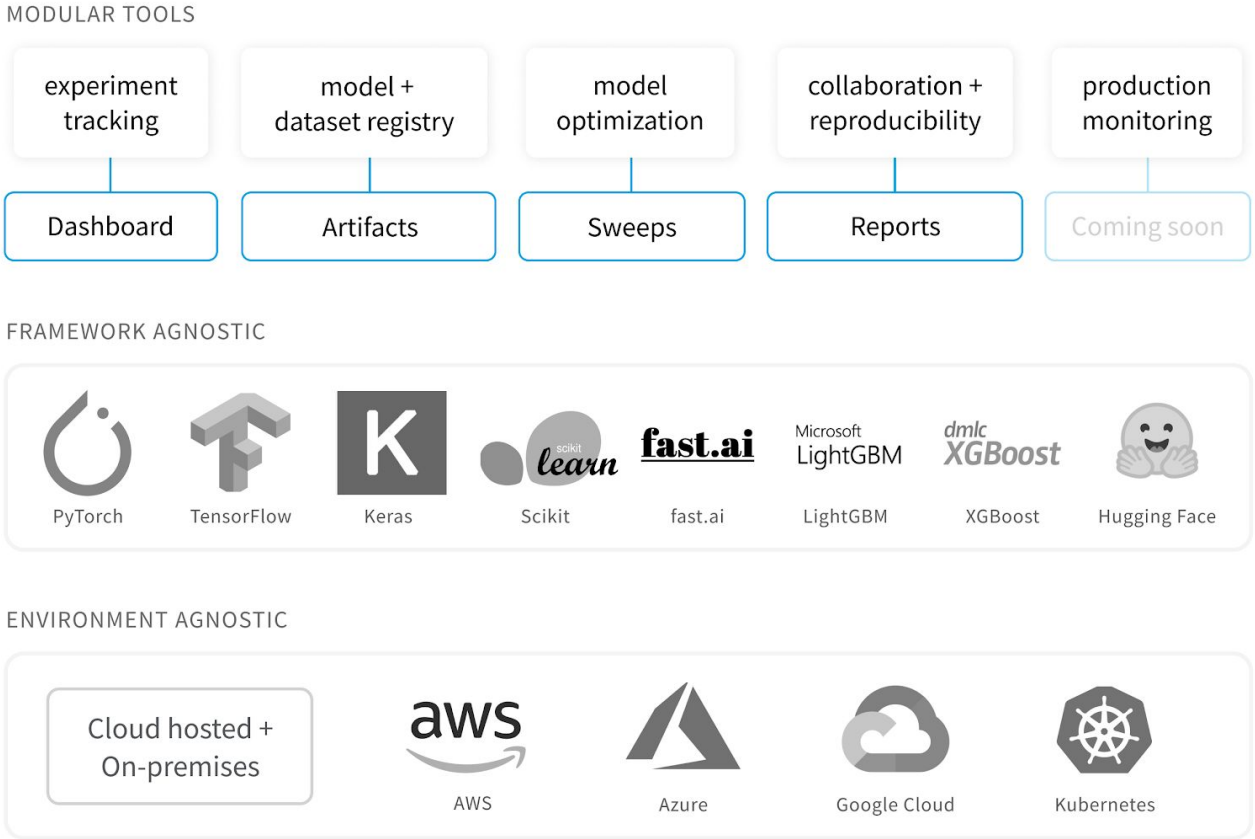


Weights & Biases

Developer tools for machine learning

https://wandb.ai/site/education

ML Developer Tools for the entire ML workflow



Light integration

Try W&B in a few minutes:

- Intro notebook https://abre.ai/wandb-hello-world
- Quickstart docs docs.wandb.com

PYTORCH

```
import torch

model.train()
```

```
import torch
import wandb

wandb.watch(model)
model.train()
```

TENSORFLOW

```
import tensorflow as tf

classifier.train()
```

```
import tensorflow as tf
import wandb

wandb.init(sync_tensorboard=True)

classifier.train()
```

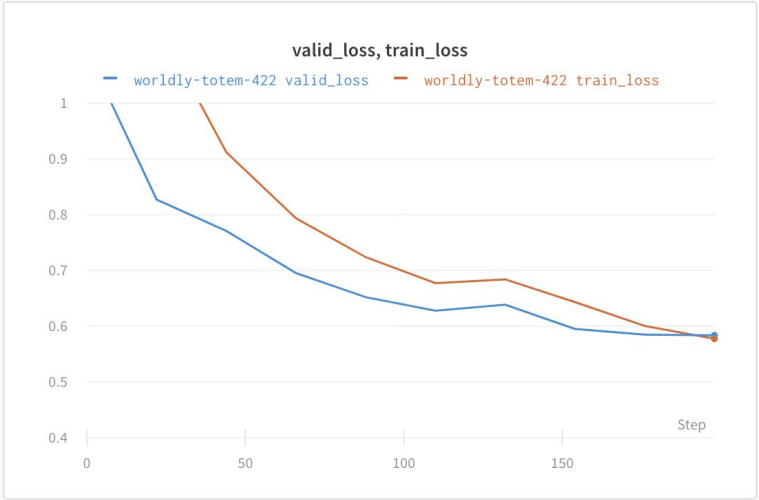
KERAS

```
from tensorflow import keras

model.fit(X, y)
```

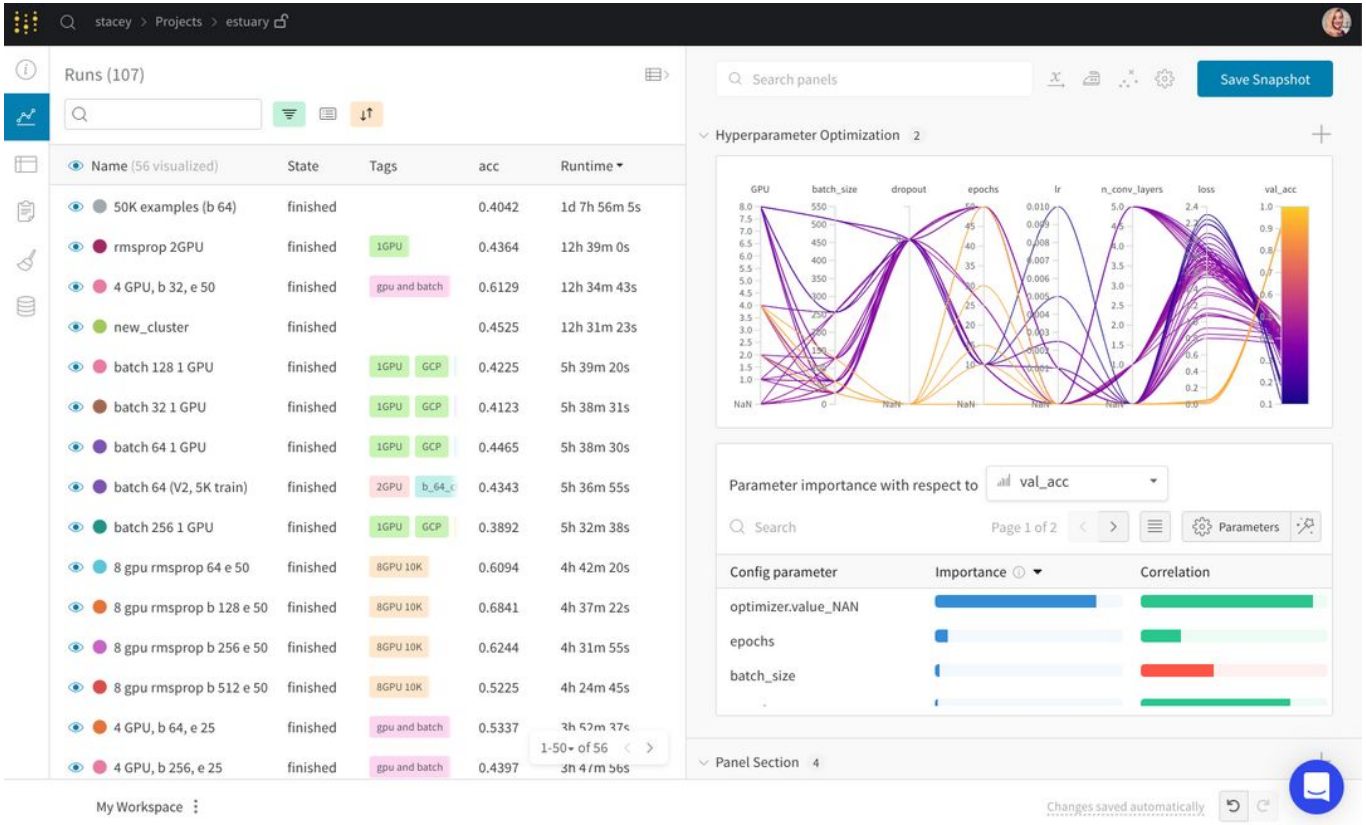
```
from tensorflow import keras
import wandb

model.fit(X, y,
callbacks=[WandbCallback()])
```

Track experiments

See live updates on model performance, check for overfitting, and visualize how a model performs on different classes.

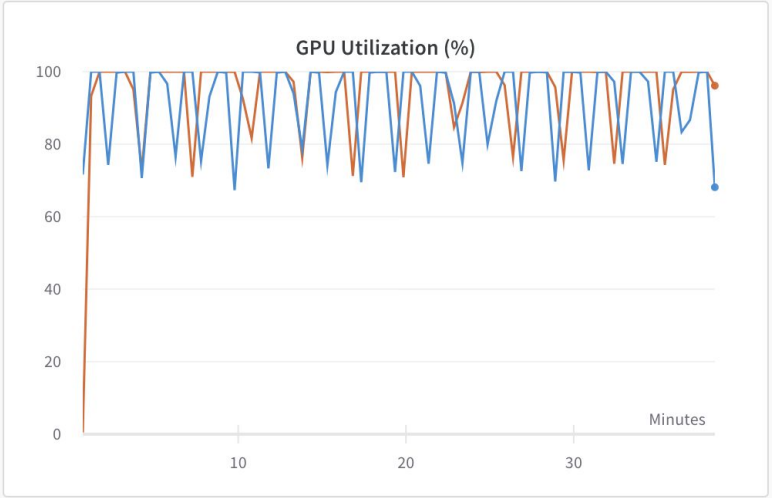


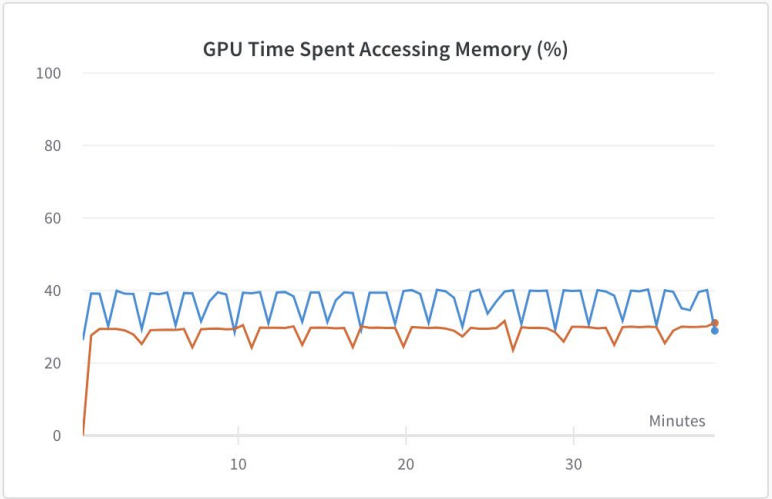
Track experiments



System metrics

Get the most out of your GPUs and identify opportunities for optimizing hardware utilization.



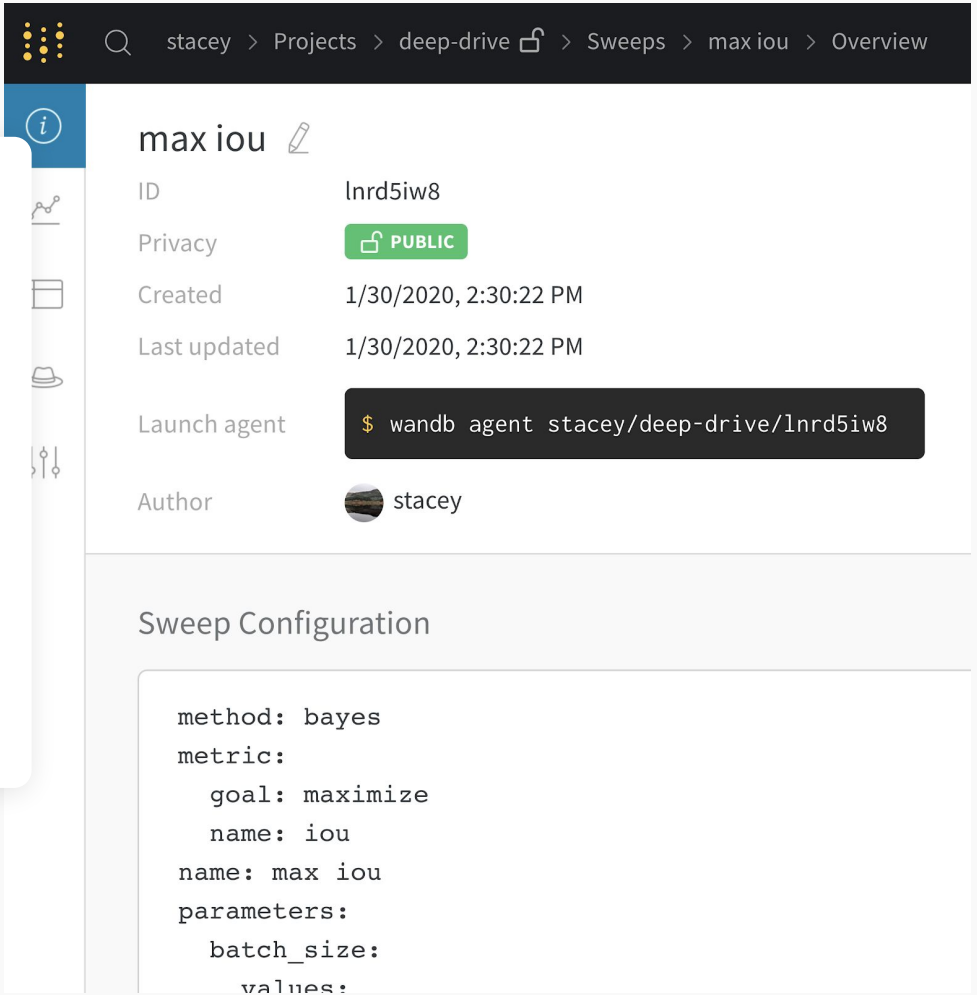


bit.ly/deep-drive-sweep Sweeps docs

HYPERPARAMETER SWEEPS

Fast setup

Use Bayesian sweeps and early stopping without writing custom code.



bit.ly/deep-drive-report

REPORTS

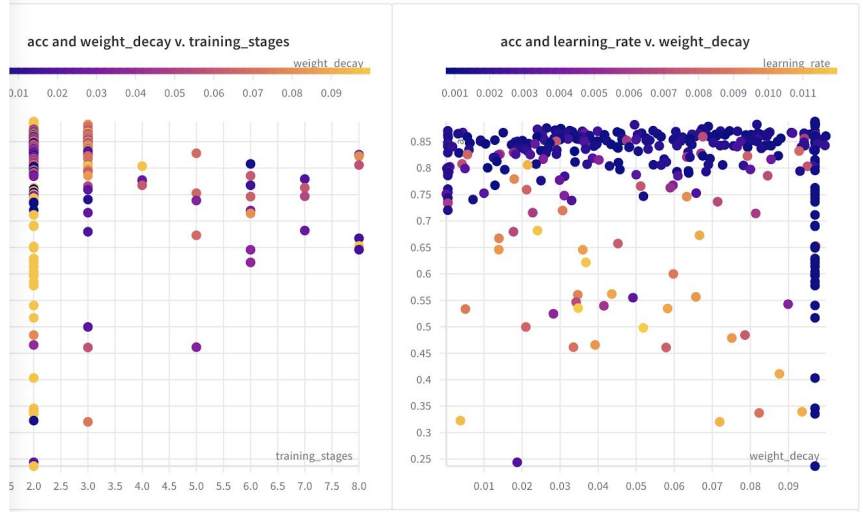
Collaborate easily

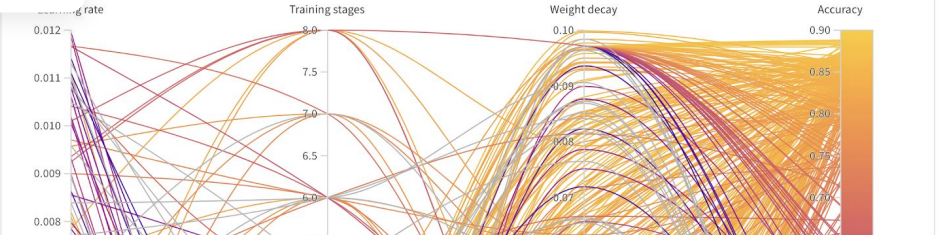
Give team members access to your exploratory results, sharing the context they need to quickly build on your work.

Weight decay: inconclusive

Initially increasing the weight decay 5X improved the accuracy by 9%. Increasing by 200X causes the same amount of improvement though.

erparameter Sweep Insights



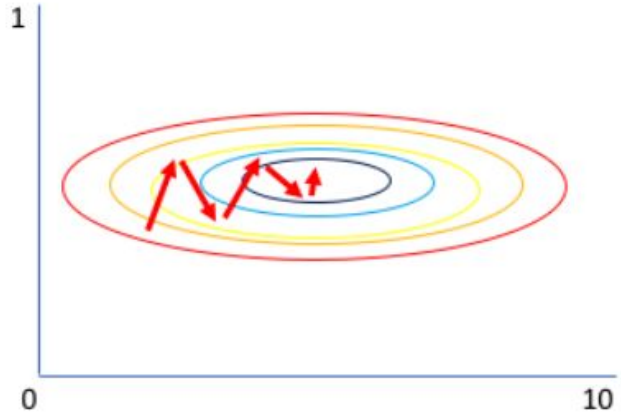


Weights & Biases

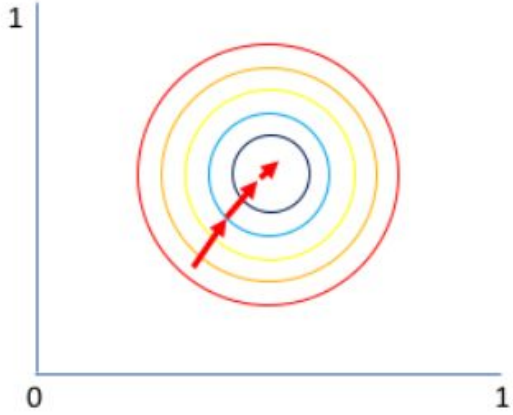
Hands On Wandb ….

Batch Normalization

Normalizing inputs to speed up learning

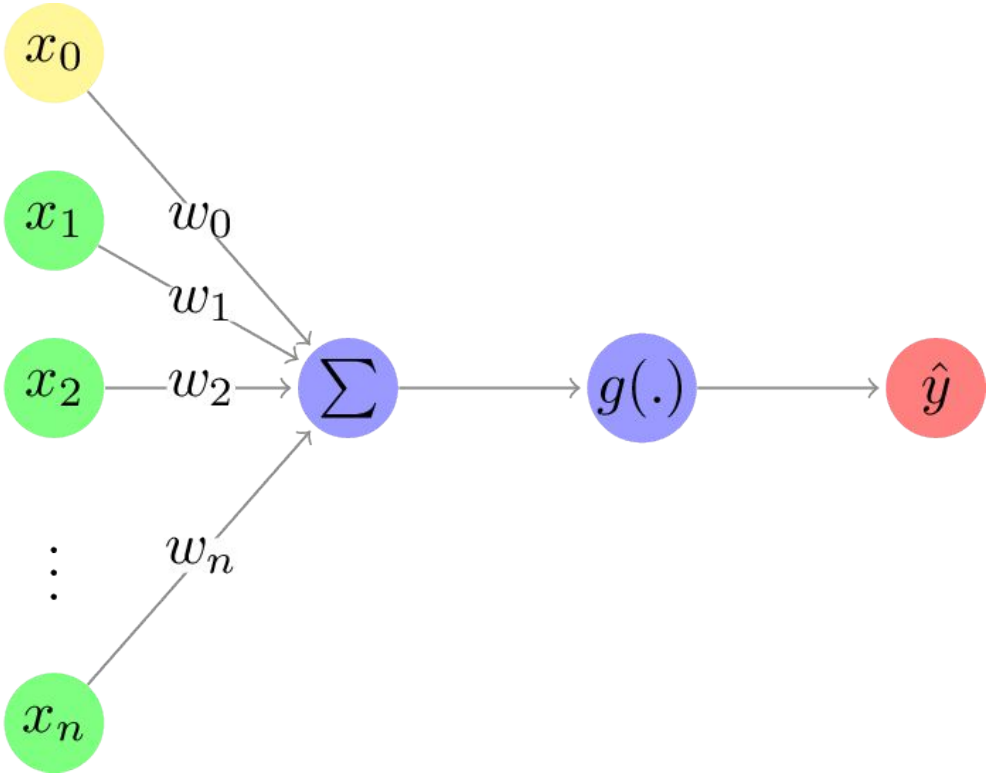


Gradient of larger parameter dominates the update



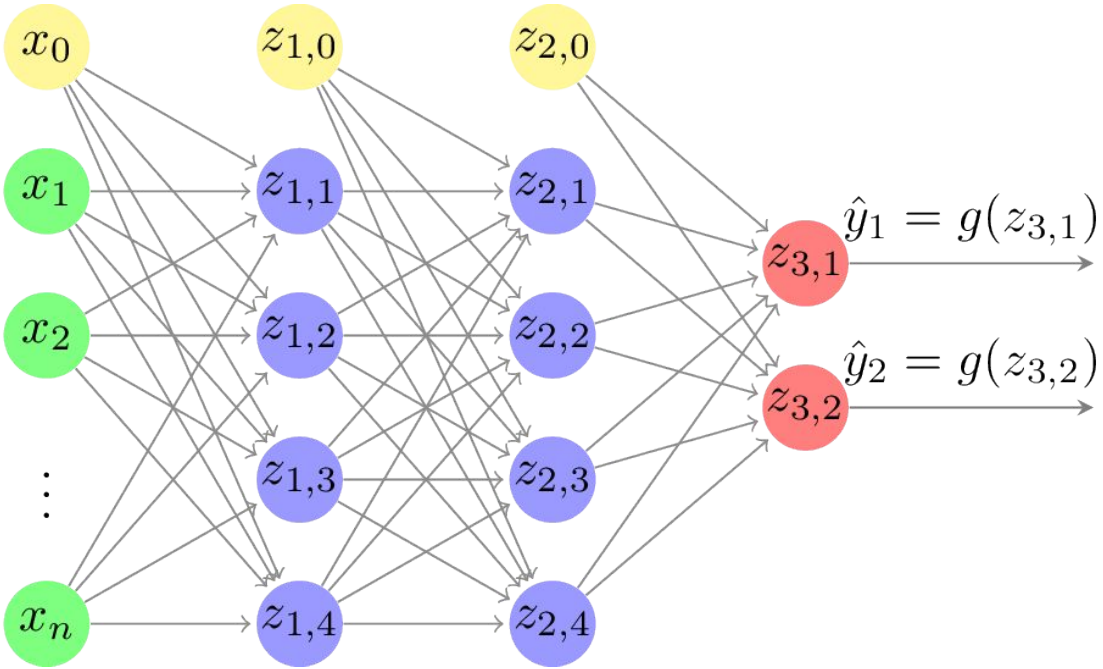
Both parameters can be updated in equal proportions

Normalizing inputs to speed up learning



$$X = \frac{X - \mu}{\sigma}$$

Normalizing inputs to speed up learning



In practice, we actually **normalize $z_{i,k}$** which has the same effect as **normalizing X.**

Batch Norm (BN)

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

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

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 initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

1 Introduction

Deep learning has dramatically advanced the state of the art in vision, speech, and many other areas. Stochastic gradient descent (SGD) has proved to be an effective way of training deep networks, and SGD variants such as momentum (Sutskever et al., 2013) and Adagrad (Duchi et al., 2011) have been used to achieve state of the art performance. SGD optimizes the parameters Θ of the network, so as to minimize the loss

$$\Theta = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, \Theta)$$

where $x_1 \ldots N$ is the training data set. With SGD, the training proceeds in steps, and at each step we consider a mini-batch $x_1 \ldots m$ of size m. The mini-batch is used to approximate the gradient of the loss function with respect to the parameters, by computing $\frac{1}{m} \frac{\partial \ell(x_i, \Theta)}{\partial \Theta}$. Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

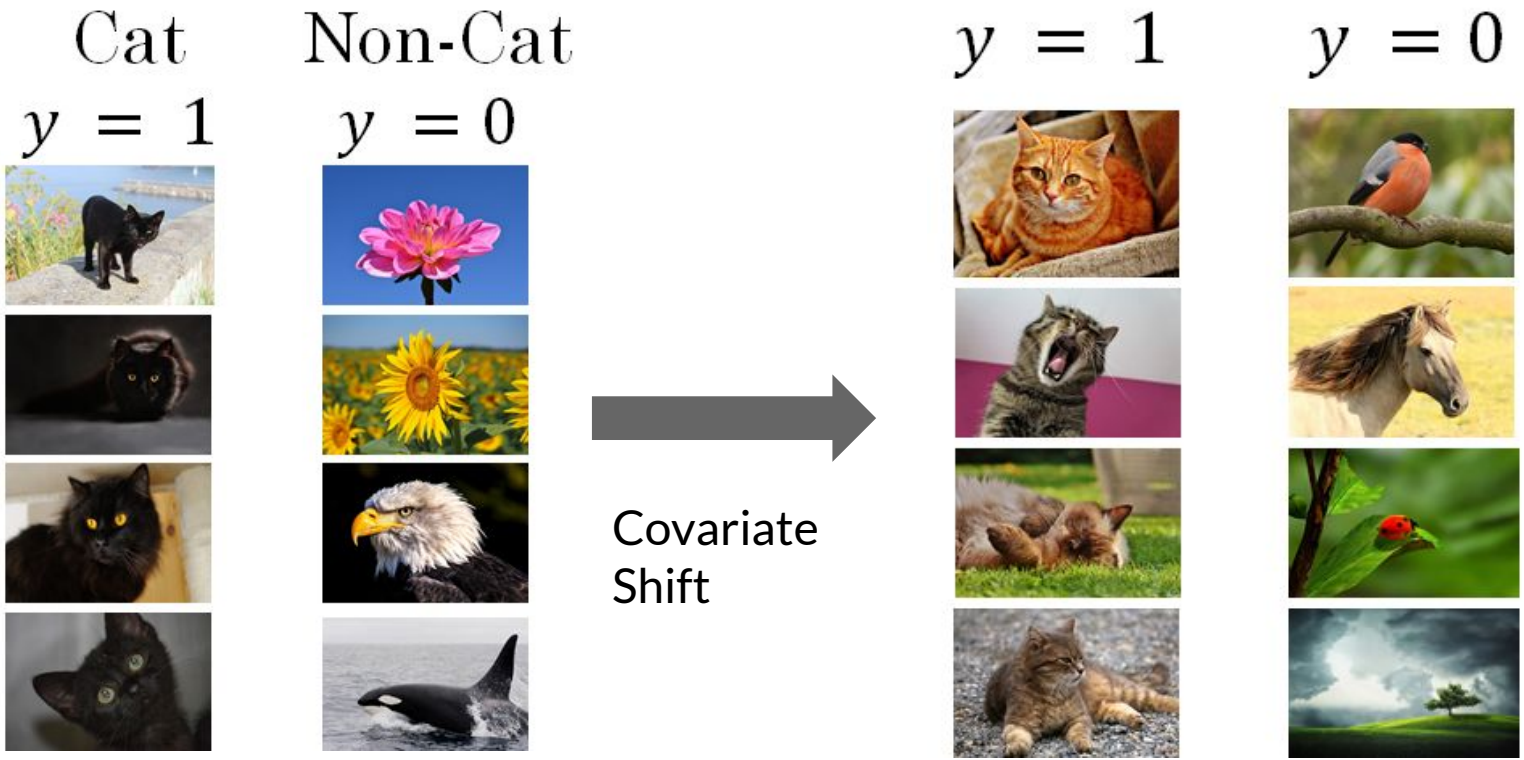
While stochastic gradient is simple and effective, it requires careful tuning of the model hyper-parameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters. The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper.

The change in the distributions of layers' inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience *covariate shift* (Shimodaira, 2000). This is typically handled via domain adaptation (Jiang, 2008). However, the notion of covariate shift can be extended beyond the learning system as a whole, to apply to its parts, such as a sub-network or a layer. Consider a network computing

$$\ell = F_2(F_1(\mathrm{u}, \Theta_1), \Theta_2)$$

where F_1 and F_2 are arbitrary transformations, and the parameters Θ_1, Θ_2 are to be learned so as to minimize

Learning on Shifting Input Distribution



How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

1 Introduction

Over the last decade, deep learning has made impressive progress on a variety of notoriously difficult tasks in computer vision [16, 7], speech recognition [5], machine translation [29], and game-playing [18, 25]. This progress hinged on a number of major advances in terms of hardware, datasets [15, 23], and algorithmic and architectural techniques [27, 12, 20, 28]. One of the most prominent examples of such advances was batch normalization (BatchNorm) [10].

At a high level, BatchNorm is a technique that aims to improve the training of neural networks by stabilizing the distributions of layer inputs. This is achieved by introducing additional network layers

Published as a conference paper at ICLR 2021

DECONSTRUCTING THE REGULARIZATION OF BATCHNORM

Yann N. Dauphin
Google Research
ynd@google.com

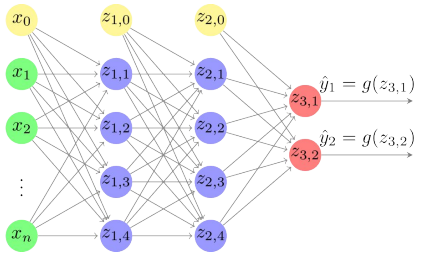
Ekin D. Cubuk
Google Research
cubuk@google.com

ABSTRACT

Batch normalization (BatchNorm) has become a standard technique in deep learning. Its popularity is in no small part due to its often positive effect on generalization. Despite this success, the regularization effect of the technique is still poorly understood. This study aims to decompose BatchNorm into separate mechanisms that are much simpler. We identify three effects of BatchNorm and assess their impact directly with ablations and interventions. Our experiments show that preventing explosive growth at the final layer at initialization and during training can recover a large part of BatchNorm's generalization boost. This regularization mechanism can lift accuracy by 2.9% for Resnet-50 on Imagenet without BatchNorm. We show it is linked to other methods like Dropout and recent initializations like Fixup. Surprisingly, this simple mechanism matches the improvement of 0.9% of the more complex Dropout regularization for the state-of-the-art Efficientnet-B8 model on Imagenet. This demonstrates the underrated effectiveness of simple regularizations and sheds light on directions to further improve generalization for deep nets.

Implementing BN in a single layer

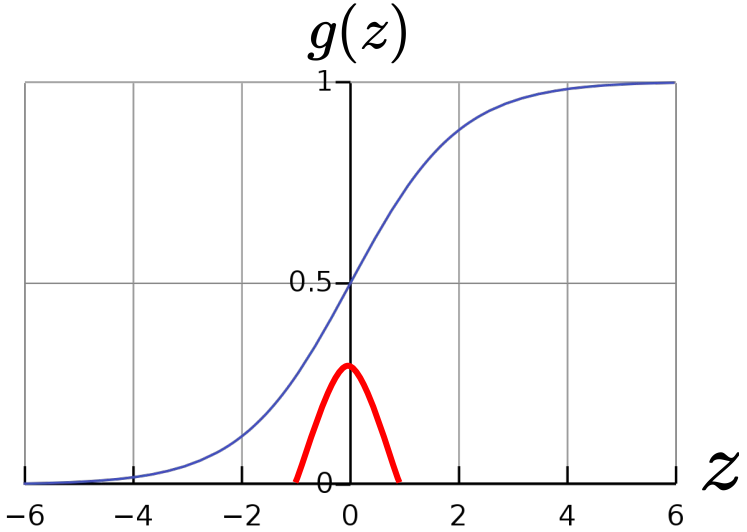
$$All \;\; z_l = \{z_l^{(1)}, z_l^{(2)}, \ldots, z_l^{(m)}\}$$



$$\mu = \frac{1}{m}\sum_i z_l^{(i)}$$

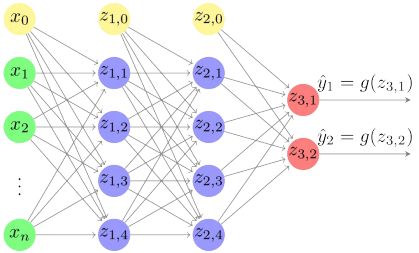
$$\sigma^2 = \frac{1}{m}\sum_i (z_l^{(i)} - \mu)^2$$

$$z_{l\_norm}^{(i)} = \frac{z_l^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$



Implementing BN in a single layer

$$All \;\; z_l = \{z_l^{(1)}, z_l^{(2)}, \ldots, z_l^{(m)}\}$$



$$\mu = \frac{1}{m}\sum_i z_l^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_l^{(i)} - \mu)^2$$

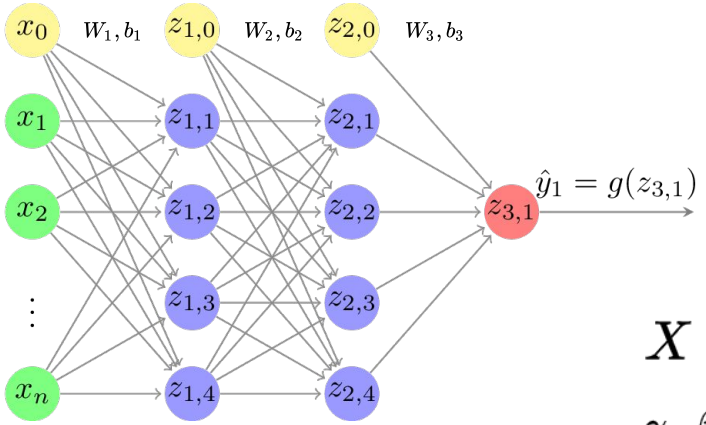
$$z_{l\_norm}^{(i)} = \frac{z_l^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z}_l^{(i)} = \gamma \, z_{l\_norm}^{(i)} + \beta$$

$$if \;\; \gamma = \sqrt{\sigma^2 + \epsilon} \;\; and \;\; \beta = \mu$$
$$z_{l\_norm}^{(i)} = z_l^{(i)}$$

- γ and β are learnable parameters (gradient descent)
- γ and β are used to avoid all layers have activations with mean zero and standard deviation equal to one.

Adding BN to a Network



$$X \xrightarrow{W_1,b_1} Z_1 \xrightarrow[\text{BN}]{\gamma_1,\beta_1} \tilde{Z}_1 \rightarrow a_1 = g_1(\tilde{Z}_1) \xrightarrow{W_2,b_2} Z_2$$

$$\xrightarrow[\text{BN}]{\gamma_2,\beta_2} \tilde{Z}_2 \rightarrow a_2 = g_2(\tilde{Z}_2) \xrightarrow{W^{[3]},b^{[3]}} \dots$$

Parameters (learnable from Gradient Descent)

$$\begin{cases} W_1, b_1, W_2, b_2, W_3, b_3 \\ \gamma_1, \beta_1, \gamma_2, \beta_2, \gamma_3, \beta_3 \end{cases}$$

Working with Mini-Batches

$$X^{\{1\}} \xrightarrow{W_1, b_1} Z_1 \xrightarrow[\text{BN}]{\gamma_1, \beta_1} \tilde{Z}_1 \rightarrow a_1 = g_1(\tilde{Z}_1) \xrightarrow{W_2, b_2} Z_2$$

$$\xrightarrow[\text{BN}]{\gamma_2, \beta_2} \tilde{Z}_2 \rightarrow a_2 = g_2(\tilde{Z}_2) \xrightarrow{W^{[3]}, b^{[3]}} \dots$$

$$\vdots$$

$$X^{\{t\}} \dots$$

BN calculates the normalization using data from individual batch t

$$\tilde{z}_l^{\{i\}} = \gamma \, z_{l\_norm}^{\{i\}} + \beta$$

Batch Normalization as Regularization

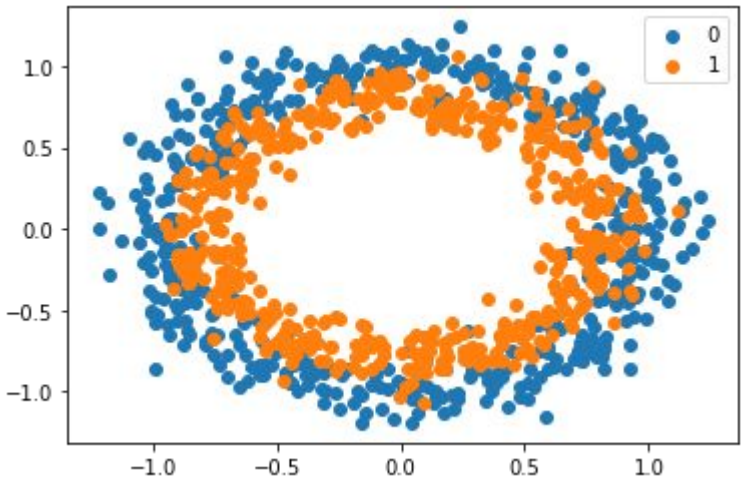
- Batch norm has **slight regularization effect** because it kind of **can cancel out large W**, adding noise.
  - Each mini-batch is scaled by the mean/standard deviation computed on just that mini-batch. This adds some noise to the values Z_l within that mini-batch. So similar to dropout, it adds some noise to each hidden layer activations.
- Its **regularization effect gets weaker as batch size gets larger** because there is less noise as batch size gets larger

```
# Batch Normalization Before Activation Function [Original Paper]
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(25, kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation(tf.nn.relu))

model.add(tf.keras.layers.Dense(12,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation(tf.nn.relu))
```

```
# Batch Normalization After Activation Function
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(25, activation=tf.nn.relu,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dense(12, activation=tf.nn.relu,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
```

```
# Batch Normalization used to standardize the raw input variables
model = tf.keras.Sequential()
model.add(BatchNormalization(input_shape=(2,)))
```

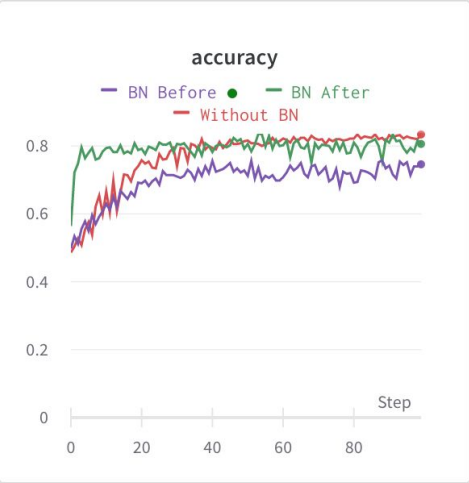
Batch Normalization - Case Study

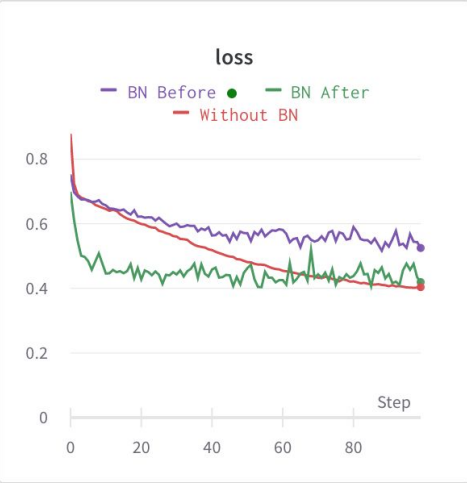


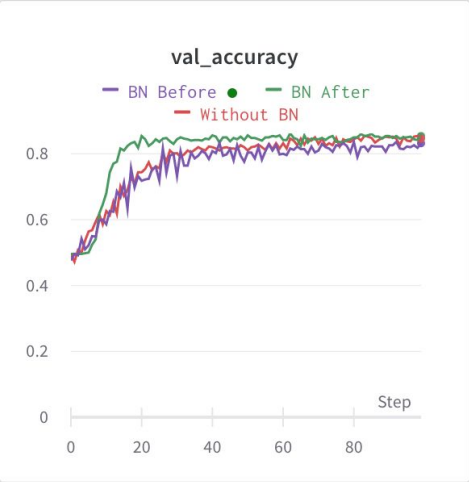

```
# Default values for hyperparameters
defaults = dict(layer_1 = 50,
                learn_rate = 0.01,
                batch_size = 32,
                epoch = 100)

wandb.init(project="lesson04_bn",
           config= defaults,
           name="lesson04_bn_run_XX")
config = wandb.config

# Define model
model = Sequential()
model.add(Dense(config.layer_1, input_dim=2,
                activation='relu',
                kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
```

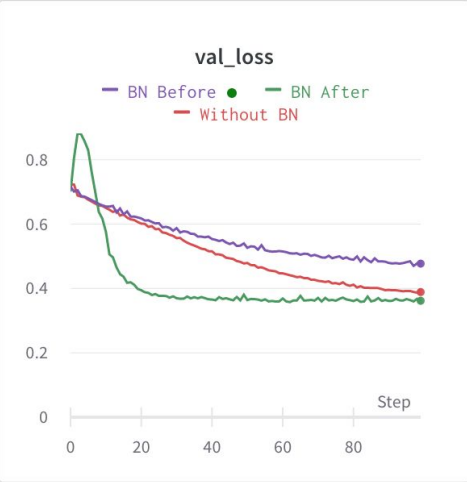
Batch Normalization - Case Study

```
# Fit model
history = model.fit(train_x, train_y,
                    validation_data=(test_x, test_y),
                    epochs=config.epoch, verbose=0,
                    batch_size=config.batch_size,
                    callbacks=[WandbCallback(log_weights=True,
                                             log_gradients=True,
                                             training_data=(train_x,train_y))])
```



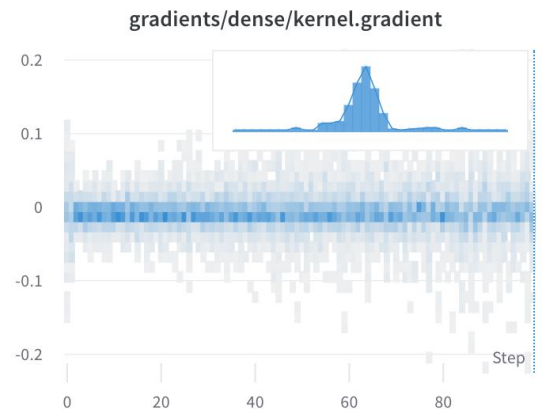


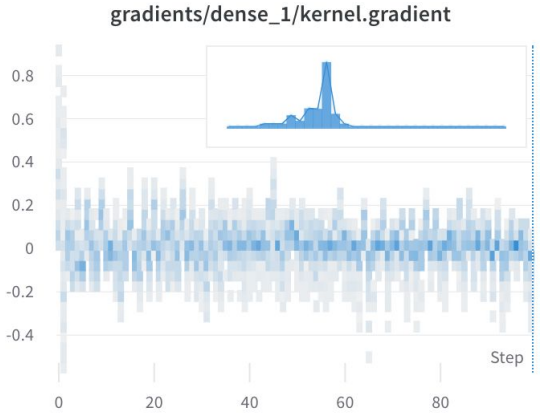




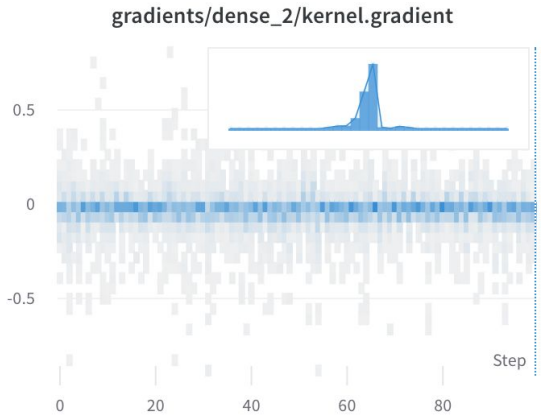
We can see that classification accuracy on the train and test datasets leap above 80% within the first 20 epochs instead of 30-to-40 epochs in the model without batch normalization.

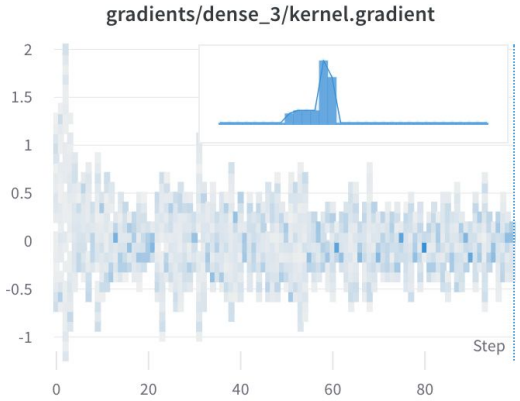
Without BN



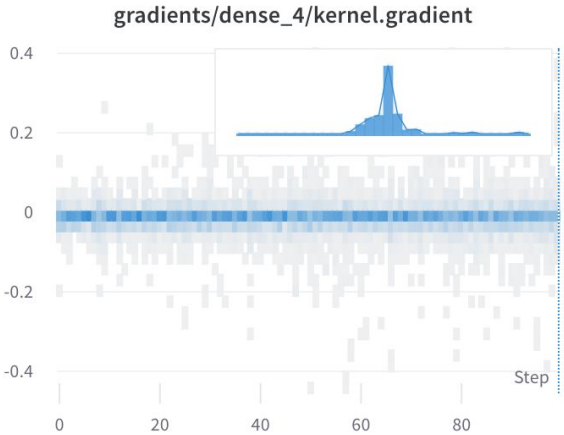


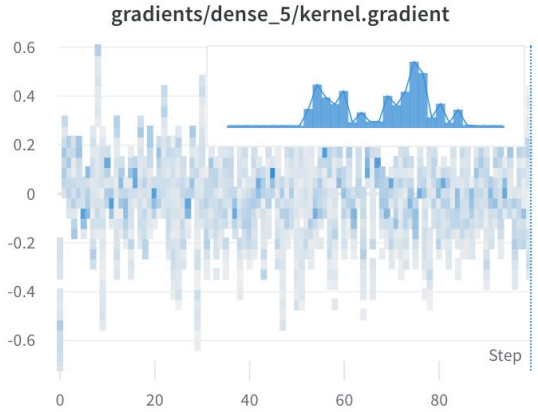
BN After





BN Before

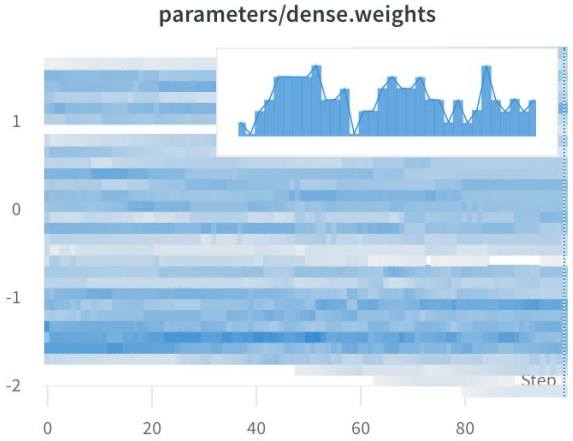


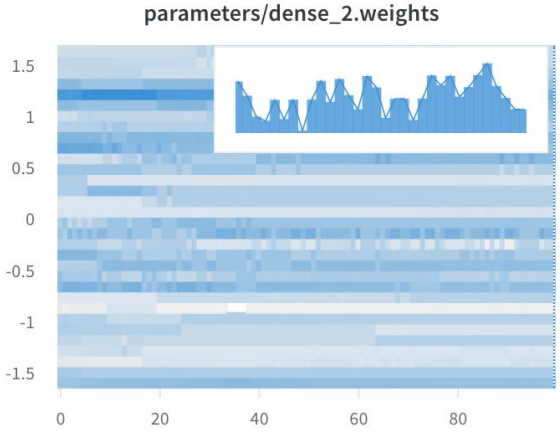


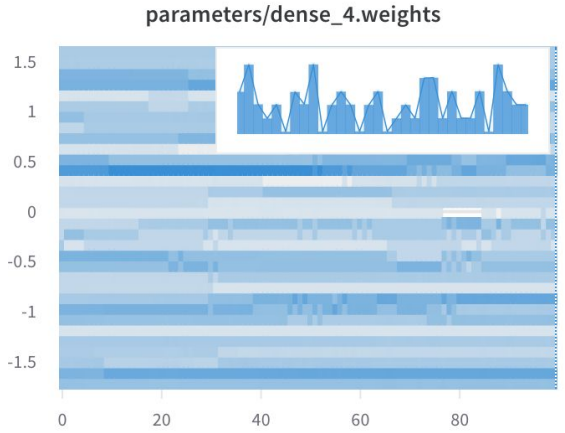
Without BN

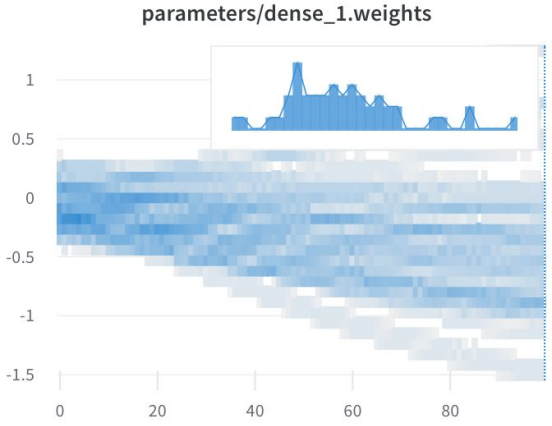
BN Before

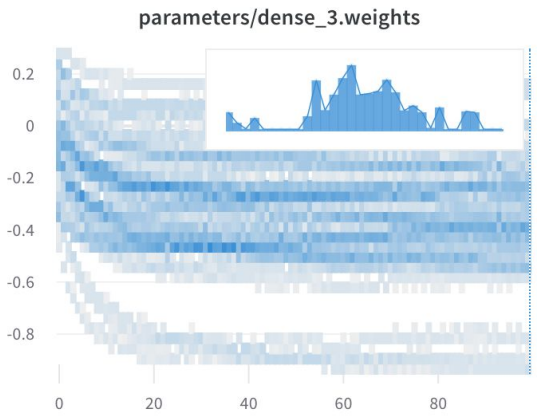
BN After

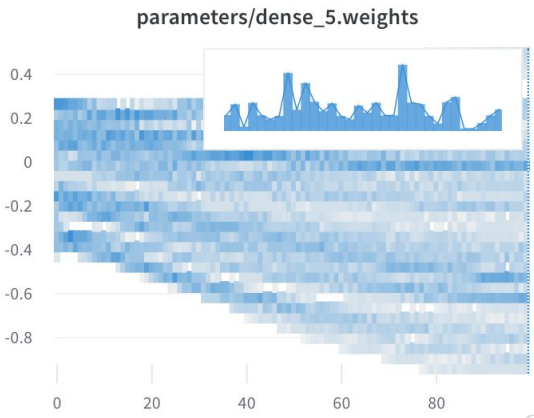






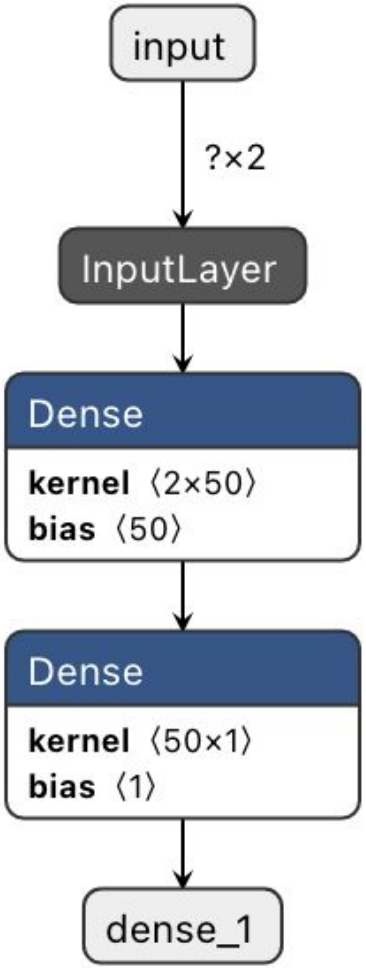






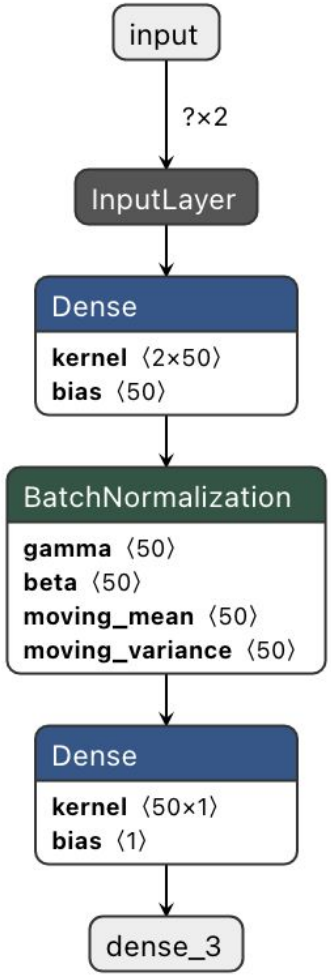
```python
# Define model
model = Sequential()
model.add(Dense(config.layer_1, input_dim=2,
                activation='relu',
                kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                150
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51
=================================================================
Total params: 201
Trainable params: 201
Non-trainable params: 0
_________________________________________________________________
```



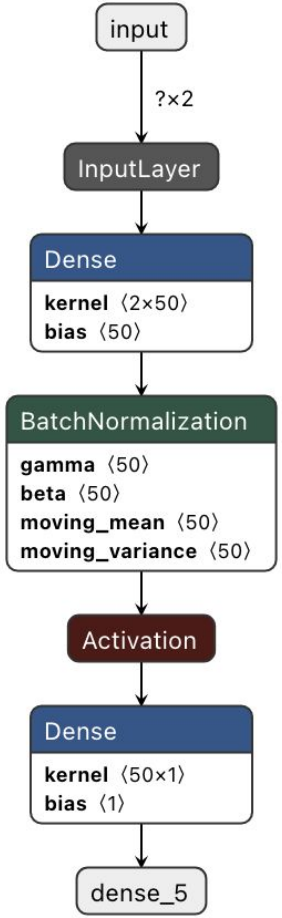
```python
# Define model
model = Sequential()
model.add(Dense(config.layer_1,
                input_dim=2,
                activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 50)                150
_________________________________________________________________
batch_normalization (BatchNo (None, 50)                200
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 51
=================================================================
Total params: 401
Trainable params: 301
Non-trainable params: 100
_________________________________________________________________
```



```python
# define model
model = Sequential()
model.add(Dense(config.layer_1, input_dim=2, kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 50)                150
_________________________________________________________________
batch_normalization_1 (Batch (None, 50)                200
_________________________________________________________________
activation (Activation)      (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 51
=================================================================
Total params: 401
Trainable params: 301
Non-trainable params: 100
_________________________________________________________________
```



Batch Normalization - Case Study

[Extensions]

- **Without Beta and Gamma**: update the example to not use the beta and gamma parameters in the batch normalization layer and compare results.
- **Without Momentum**: update the example not to use momentum in the batch normalization layer during training and compare results.
- **Input Layer**: update the example to use batch normalization after the input to the model and compare results.