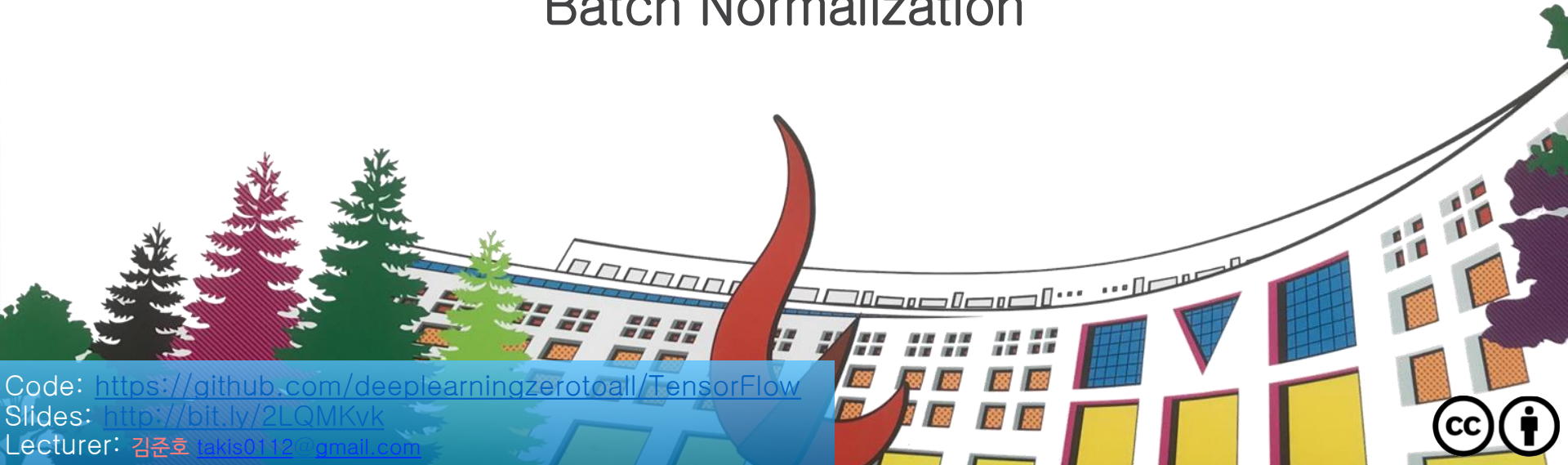
ML/DL for Everyone Season2

with TensorFlow

Lab10-4
Batch Normalization



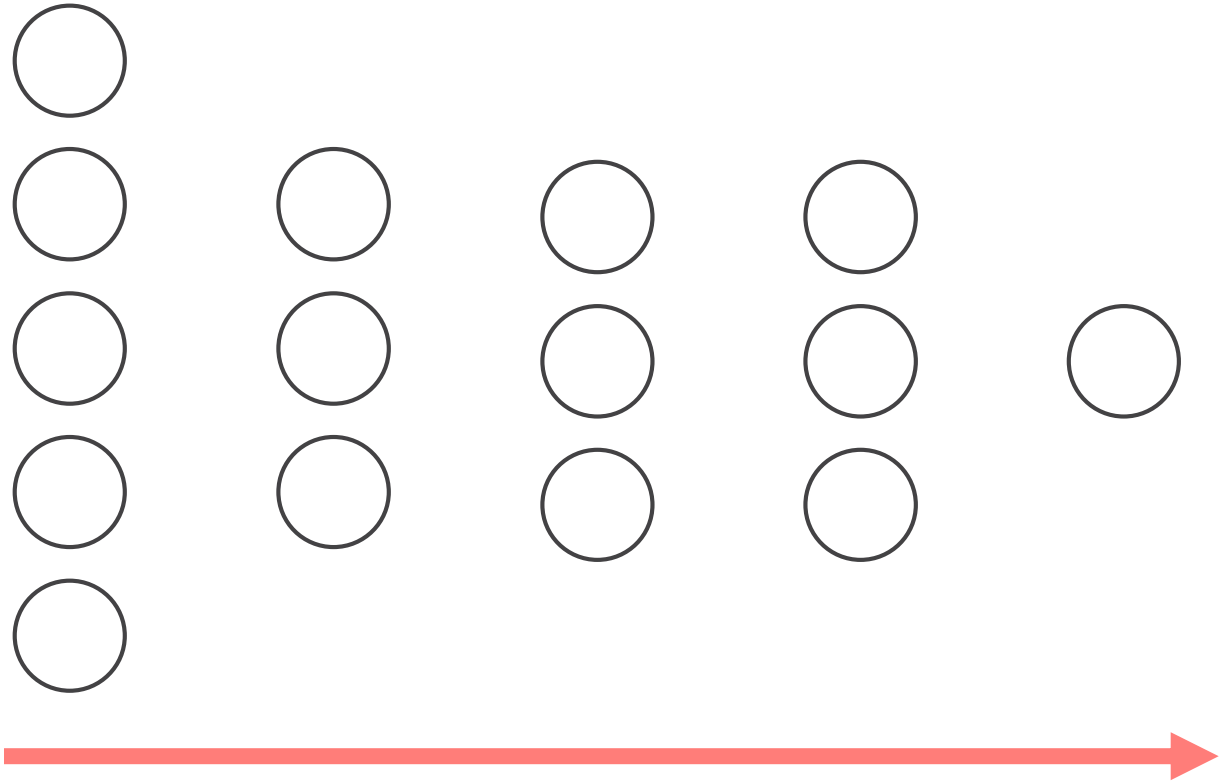
Code: https://github.com/deeplearningzerotoall/TensorFlow
Slides: http://bit.ly/2LQMKvk
Lecturer: 김준호 taki0112@gmail.com

Lab10-4: Batch Normalization

- Batch Normalization
- Code
  - load dataset
  - create network
  - define loss function
  - experiments
    - parameters
    - model
    - eager mode
- What's Next

Batch Normalization

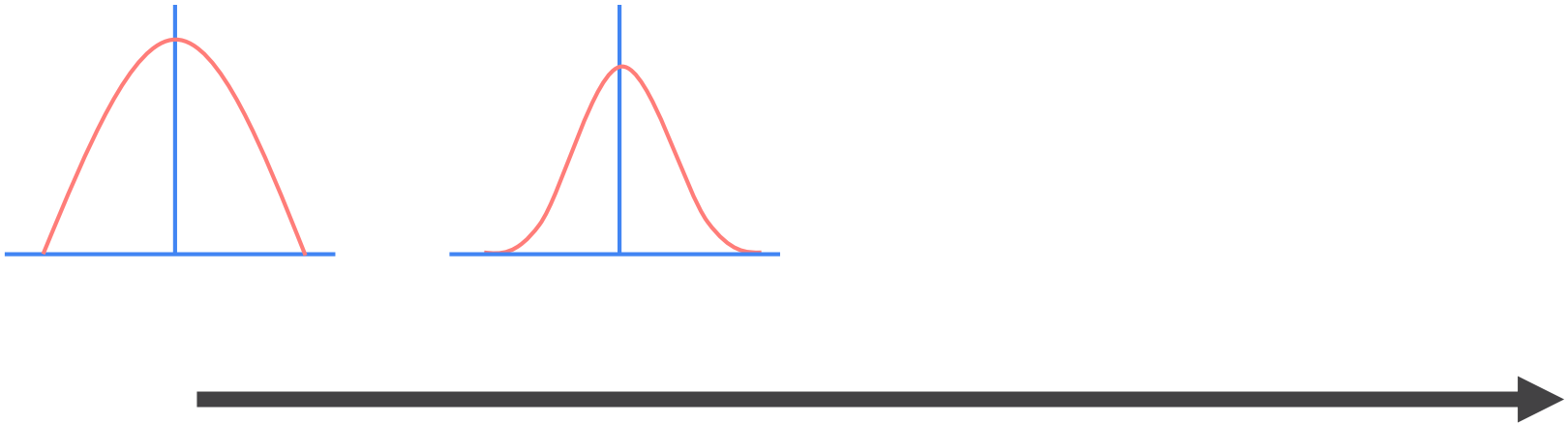
Batch Normalization



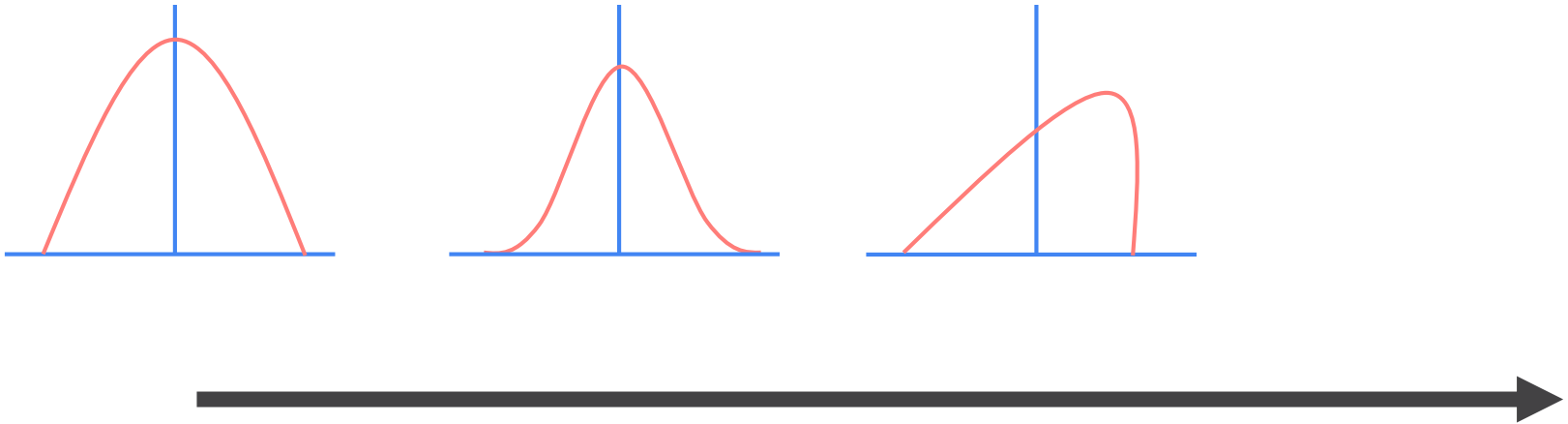
Cat

Batch Normalization

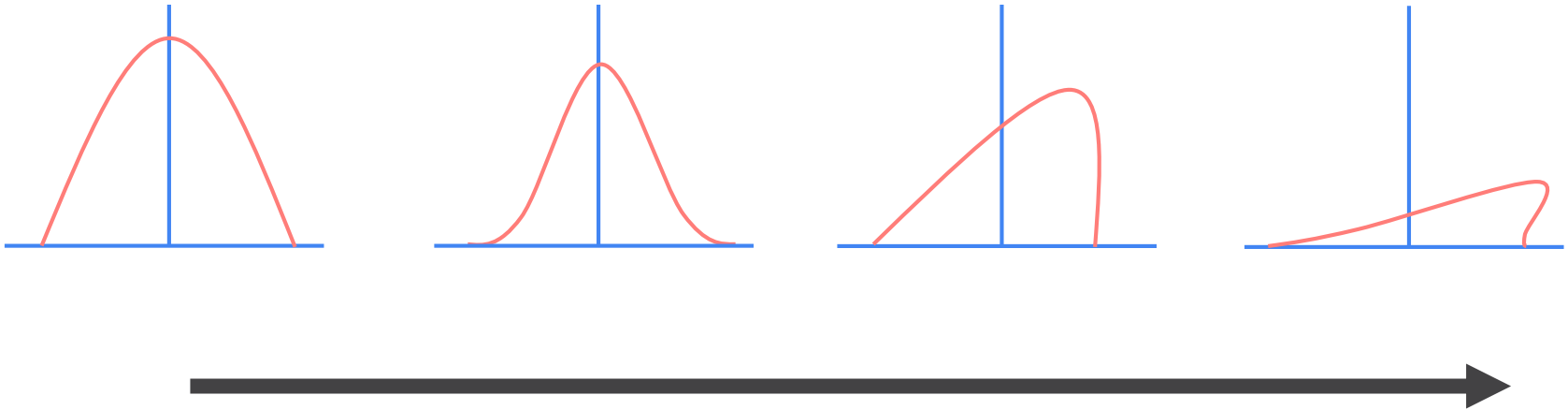
Batch Normalization



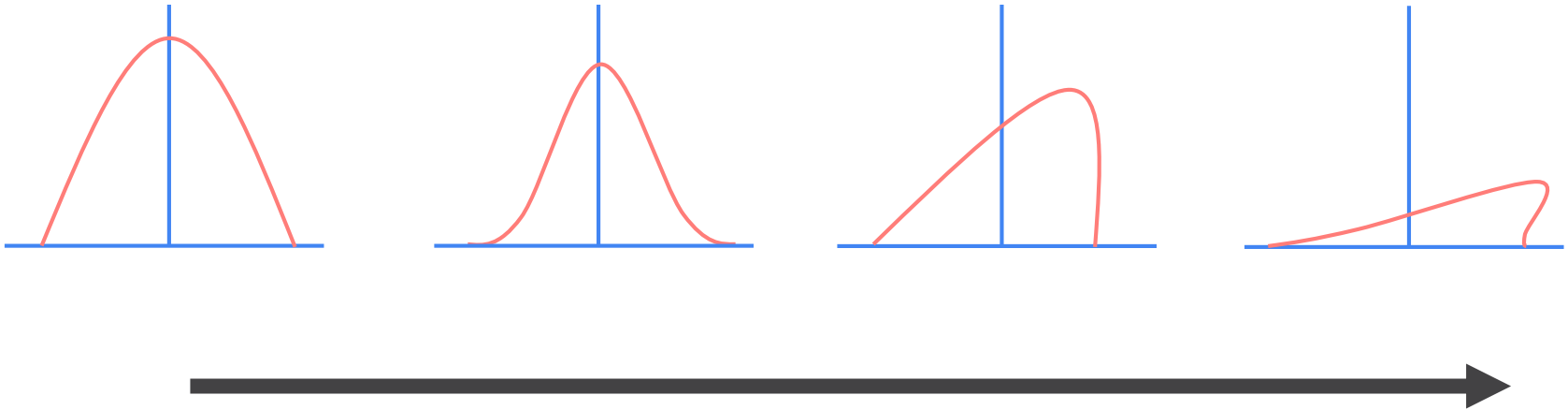
Batch Normalization



Batch Normalization

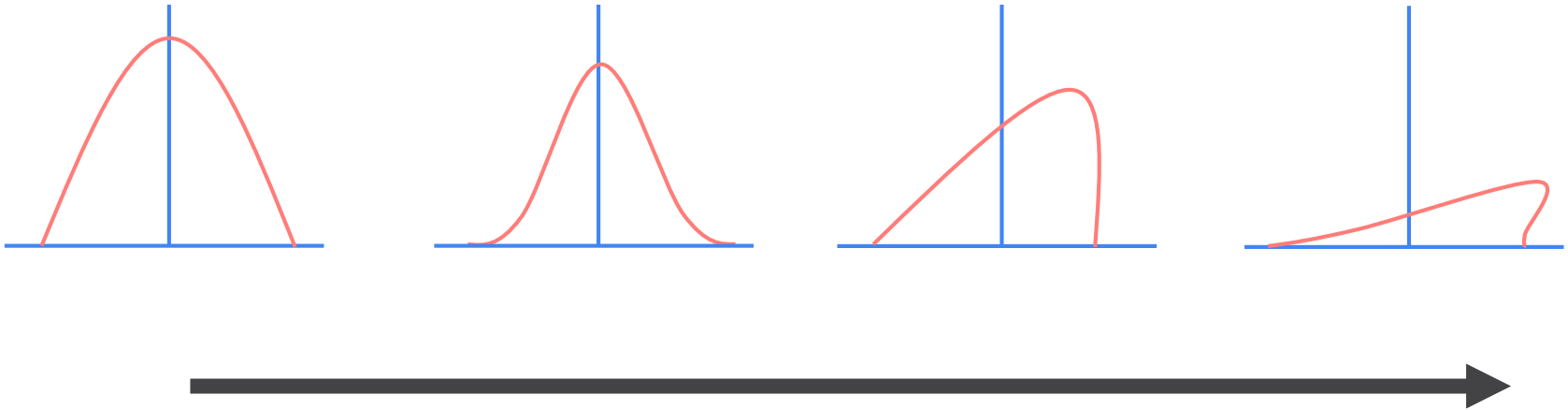


Batch Normalization



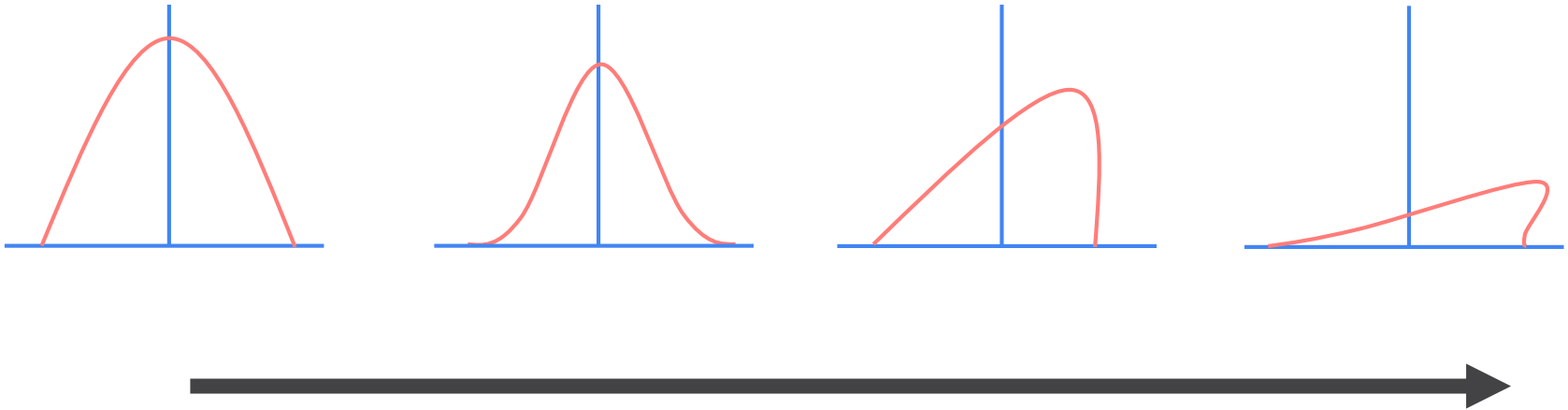
Internal Covariate Shift

Batch Normalization



$$\overline{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

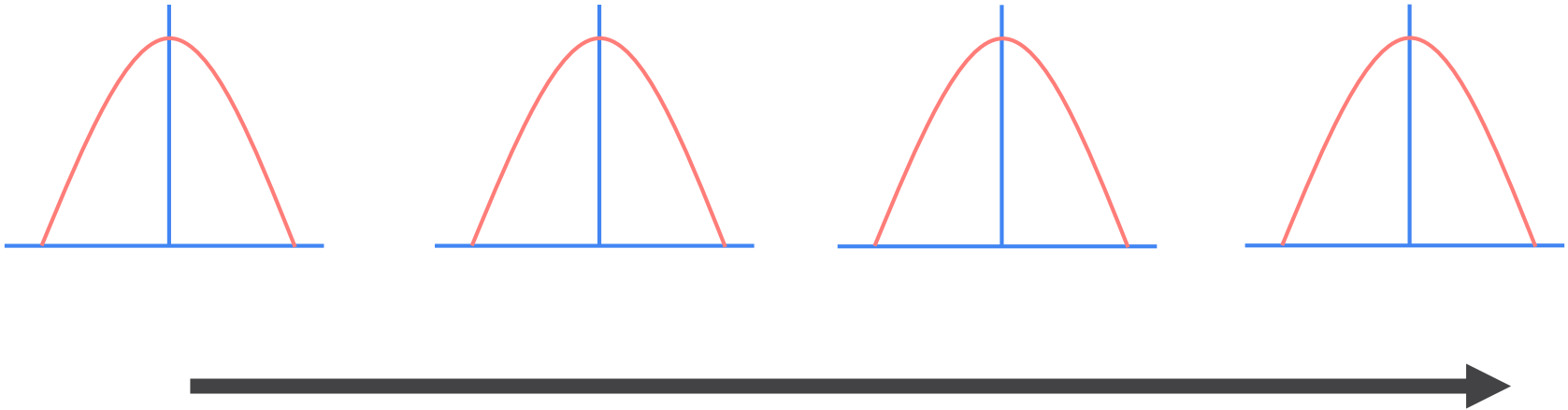
Batch Normalization



$$\overline{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{x} = \gamma\bar{x} + \beta$$

Batch Normalization



$$\overline{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{x} = \gamma \bar{x} + \beta$$

Code

Load mnist

Load mnist

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist # fasion_mnist, cifar10, cifar100
tf.enable_eager_execution()

def load_mnist() :
    (train_data, train_labels), (test_data, test_labels) = mnist.load_data()

    train_data = np.expand_dims(train_data, axis=-1) # [N, 28, 28] -> [N, 28, 28, 1]
    test_data = np.expand_dims(test_data, axis=-1) # [N, 28, 28] -> [N, 28, 28, 1]

    train_data, test_data = normalize(train_data, test_data) # [0 ~ 255] -> [0 ~ 1]

    train_labels = to_categorical(train_labels, 10) # [N,] -> [N, 10]
    test_labels = to_categorical(test_labels, 10) # [N,] -> [N, 10]

    return train_data, train_labels, test_data, test_labels


  def normalize(train_data, test_data):
      train_data = train_data.astype(np.float32) / 255.0
      test_data = test_data.astype(np.float32) / 255.0

      return train_data, test_data
```

Create network

Create network

```
def flatten() :
    return tf.keras.layers.Flatten()

def dense(channel, weight_init) :
    return tf.keras.layers.Dense(units=channel, use_bias=True, kernel_initializer=weight_init)

def relu() :
    return tf.keras.layers.Activation(tf.keras.activations.relu)

def dropout(rate) :
    return tf.keras.layers.Dropout(rate)
```

Create network

```
def flatten() :
    return tf.keras.layers.Flatten()

def dense(channel, weight_init) :
    return tf.keras.layers.Dense(units=channel, use_bias=True, kernel_initializer=weight_init)

def relu() :
    return tf.keras.layers.Activation(tf.keras.activations.relu)

def batch_norm() :
    return tf.keras.layers.BatchNormalization()
```

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(relu())
            self.model.add(dropout(rate=0.5))

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(relu())
            self.model.add(dropout(rate=0.5))

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(batch_norm())
            self.model.add(relu())

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(batch_norm())
            self.model.add(relu())

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(batch_norm())
            self.model.add(relu())

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

layer
norm
activation

Create network

```
class create_model(tf.keras.Model):
    def __init__(self, label_dim):
        super(create_model, self).__init__()

        weight_init = tf.keras.initializers.glorot_uniform()
        self.model = tf.keras.Sequential()

        self.model.add(flatten()) # [N, 28, 28, 1] -> [N, 784]

        for i in range(2):
            # [N, 784] -> [N, 256] -> [N, 256]
            self.model.add(dense(256, weight_init))
            self.model.add(batch_norm())
            self.model.add(relu())

        self.model.add(dense(label_dim, weight_init)) # [N, 256] -> [N, 10]

    def call(self, x, training=None, mask=None):
        x = self.model(x)

        return x
```

layer
norm
activation

norm
activation
layer

Create network

```
def create_model(label_dim) :

    weight_init = tf.keras.initializers.glorot_uniform()

    model = tf.keras.Sequential()
    model.add(flatten())

    for i in range(2) :
        model.add(dense(256, weight_init))
        model.add(batch_norm())
        model.add(relu())

    model.add(dense(label_dim, weight_init))

    return model
```

Define loss

Define loss

```
def loss_fn(model, images, labels):
    logits = model(images, training=True)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=labels))
    return loss


def accuracy_fn(model, images, labels):
    logits = model(images, training=False)
    prediction = tf.equal(tf.argmax(logits, -1), tf.argmax(labels, -1))
    accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32))
    return accuracy



def grad(model, images, labels):
    with tf.GradientTape() as tape:
        loss = loss_fn(model, images, labels)
    return tape.gradient(loss, model.variables)
```

Experiments (parameters)

Experiments (parameters)

```
""" parameters """
learning_rate = 0.001
batch_size = 128

training_epochs = 1
training_iterations = len(train_x) // batch_size

label_dim = 10
```

```
""" dataset """
train_x, train_y, test_x, test_y = load_mnist()
```

```
""" Graph Input using Dataset API """
train_dataset = tf.data.Dataset.from_tensor_slices((train_x, train_y)).\
    shuffle(buffer_size=100000).\
    prefetch(buffer_size=batch_size).\
    batch(batch_size).\
    repeat()

test_dataset = tf.data.Dataset.from_tensor_slices((test_x, test_y)).\
    shuffle(buffer_size=100000).\
    prefetch(buffer_size=len(test_x)).\
    batch(len(test_x)).\
    repeat()
```

Experiments (model)

Experiments (model)

```
""" Dataset Iterator """
train_iterator = train_dataset.make_one_shot_iterator()
test_iterator = test_dataset.make_one_shot_iterator()
```

```
""" Model """
network = create_model(label_dim)
```

```
""" Training """
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Experiments (Eager mode)

Experiments (Eager mode)

```
checkpoint = tf.train.Checkpoint(dnn=network)
global_step = tf.train.create_global_step()

for epoch in range(start_epoch, training_epochs):
    for idx in range(start_iteration, training_iterations):
        train_input, train_label = train_iterator.get_next()

        grads = grad(network, train_input, train_label)
        optimizer.apply_gradients(grads_and_vars=zip(grads, network.variables), global_step=global_step)

        train_loss = loss_fn(network, train_input, train_label)
        train_accuracy = accuracy_fn(network, train_input, train_label)

        test_input, test_label = test_iterator.get_next()
        test_accuracy = accuracy_fn(network, test_input, test_label)

        print("Epoch: [%2d] [%5d/%5d], train_loss: %.8f, train_accuracy: %.4f, test_Accuracy: %.4f" \
            % (epoch, idx, training_iterations, train_loss, train_accuracy, test_accuracy))
        counter += 1

checkpoint.save(file_prefix=checkpoint_prefix + '-{}'.format(counter))
```

What’s Next?

- Convolution Neural Networks (CNN)