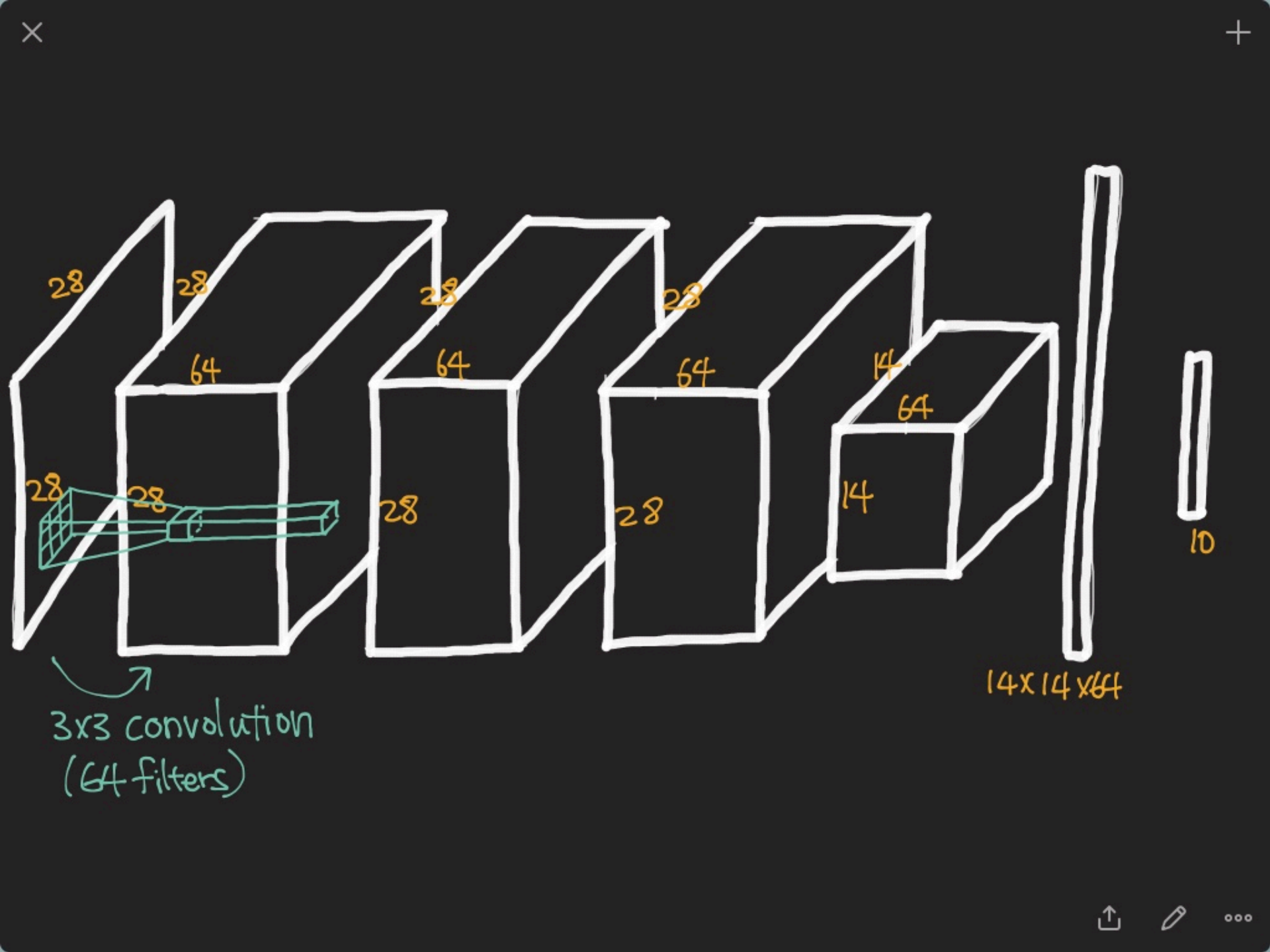
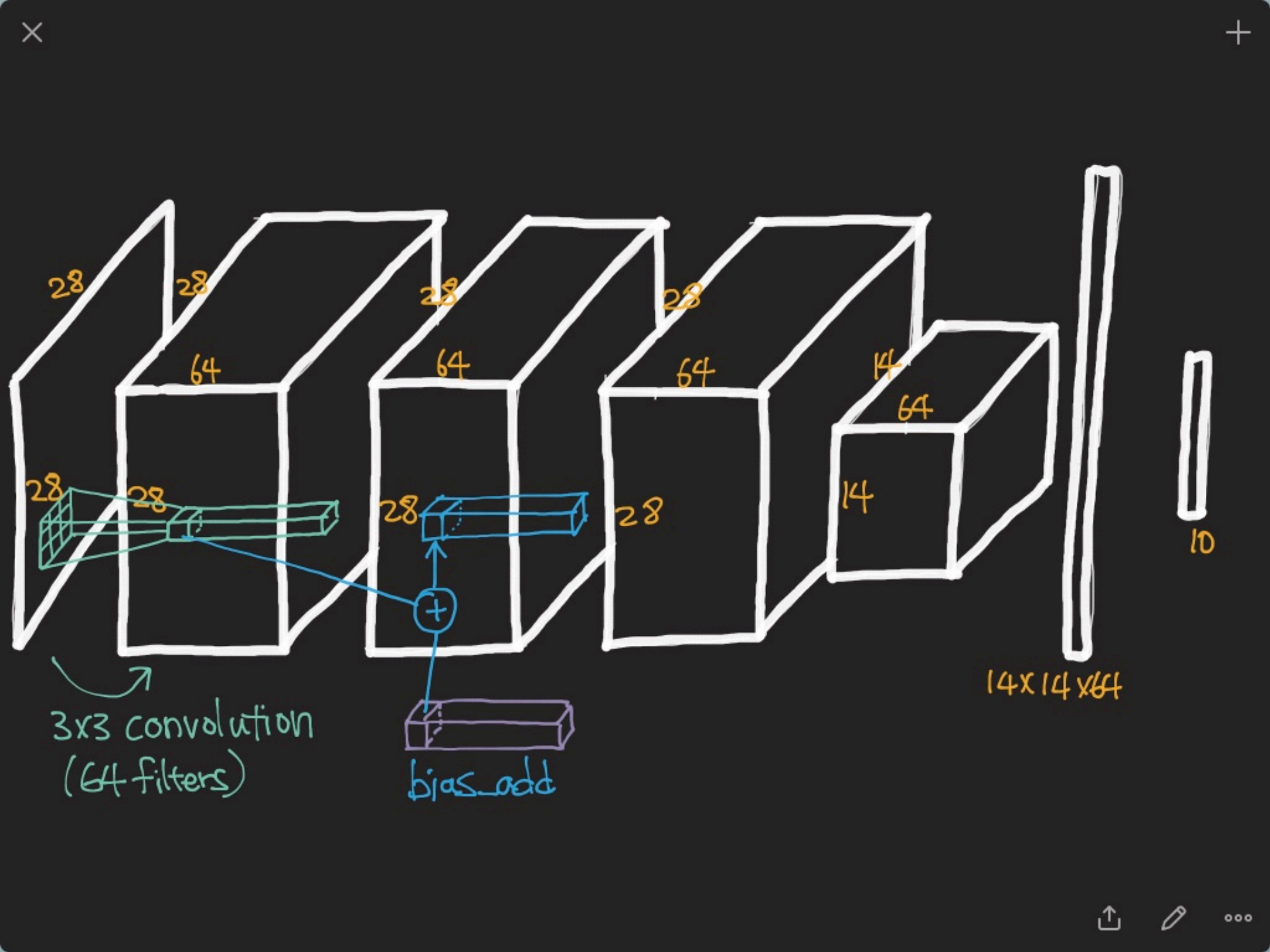
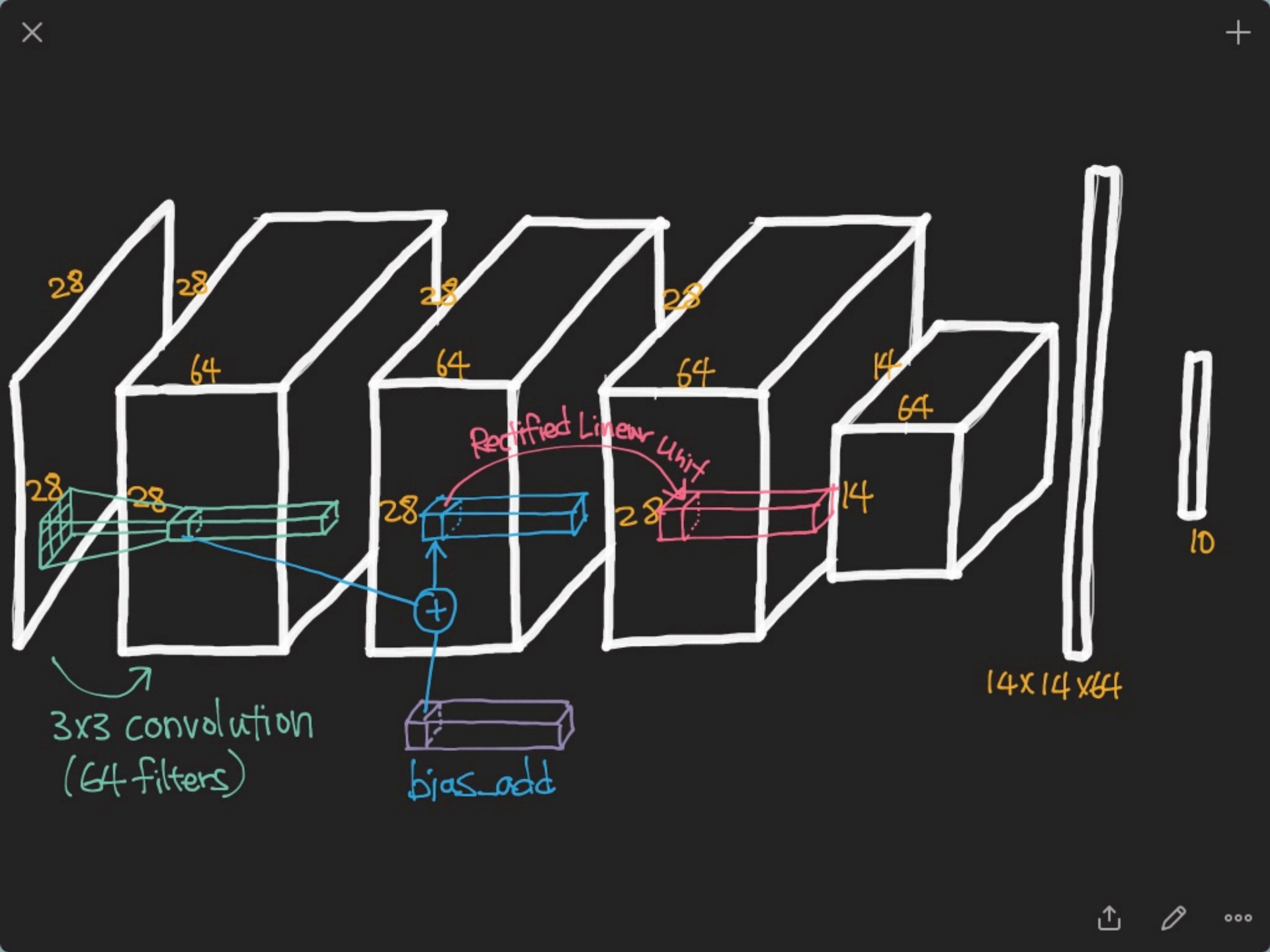
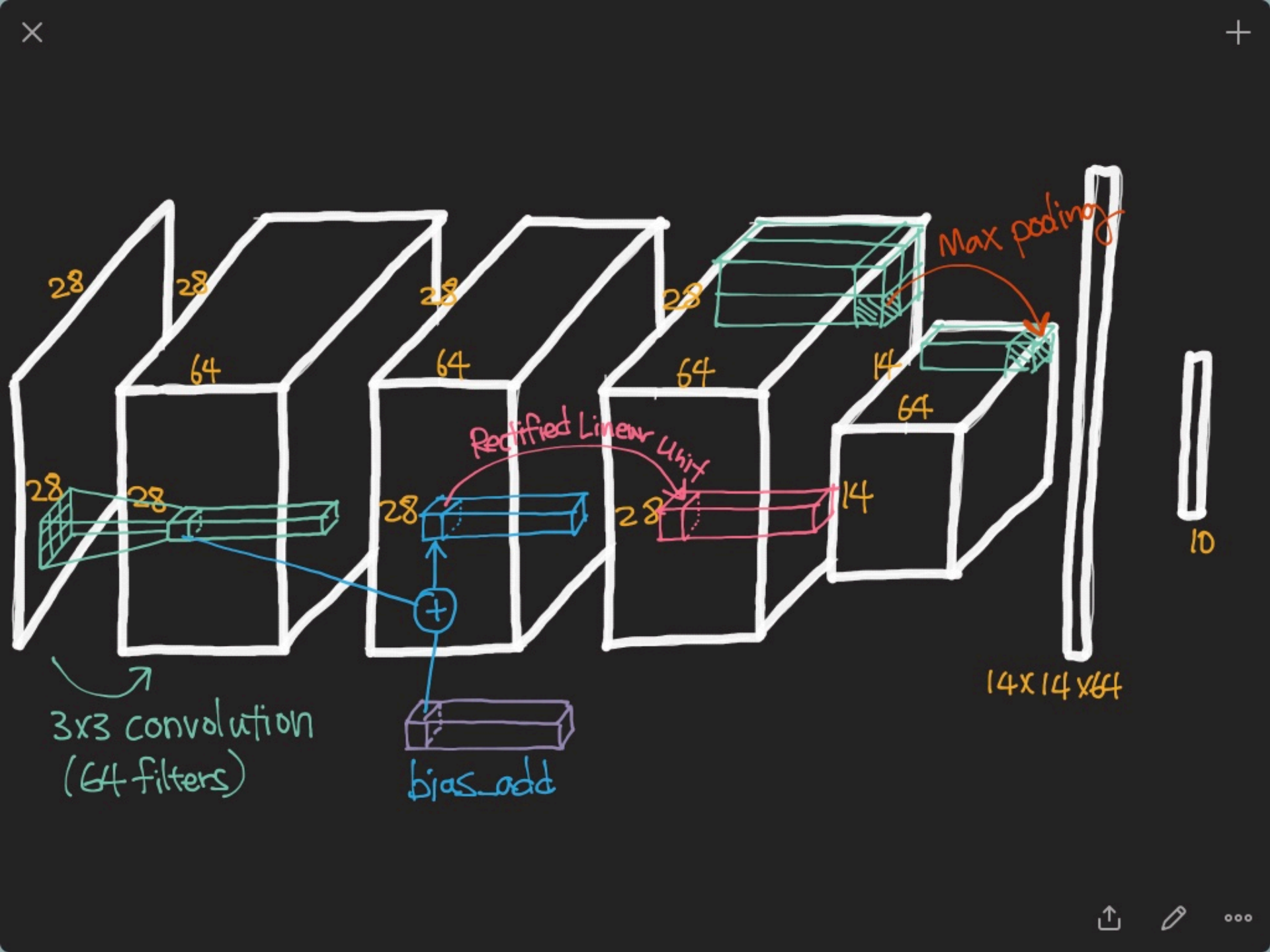
Implementing convolutional neural networks

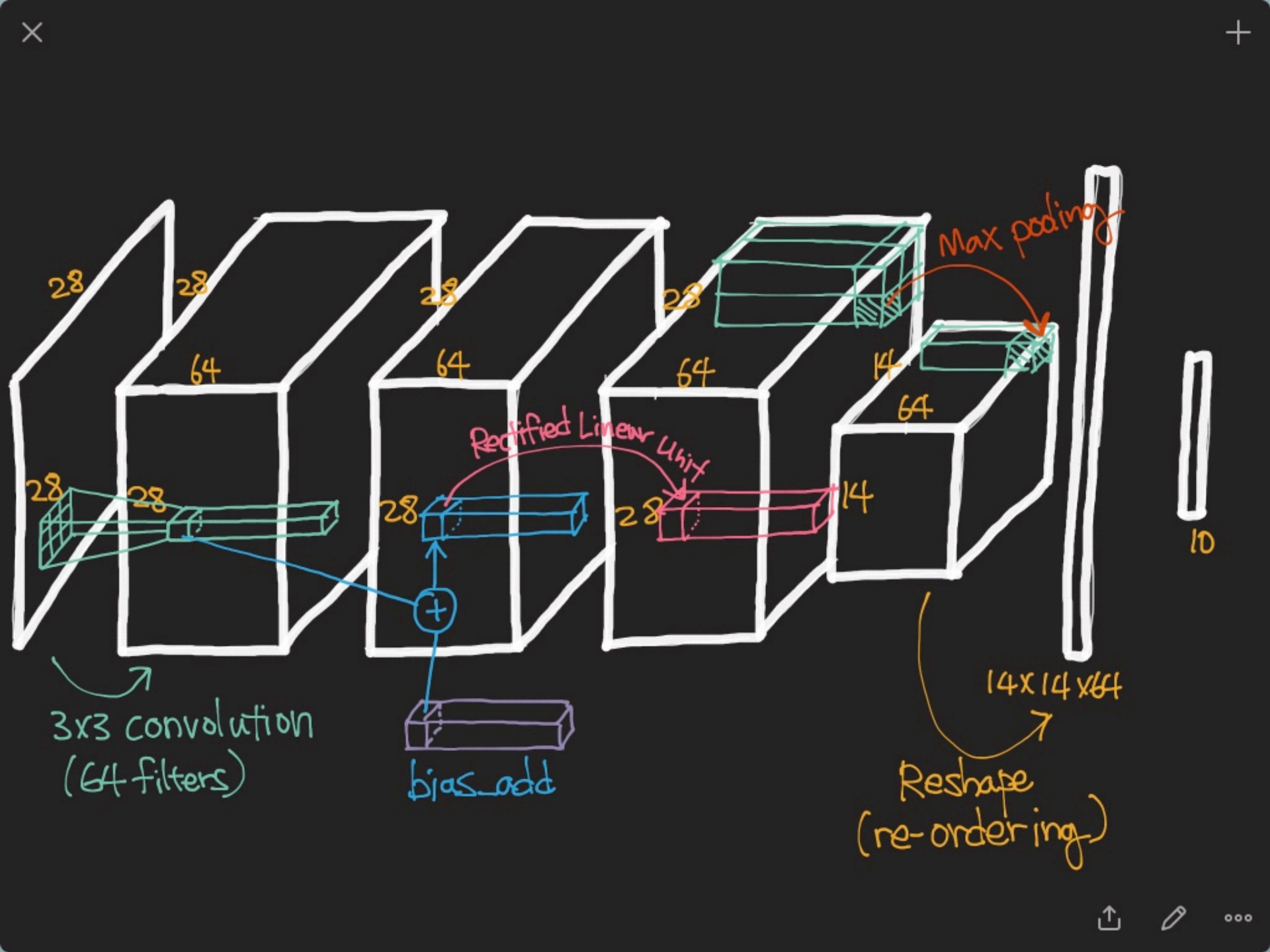
28
28
28
28
64
64
64
14
64
28
28
28
28
14
14X14X64
10

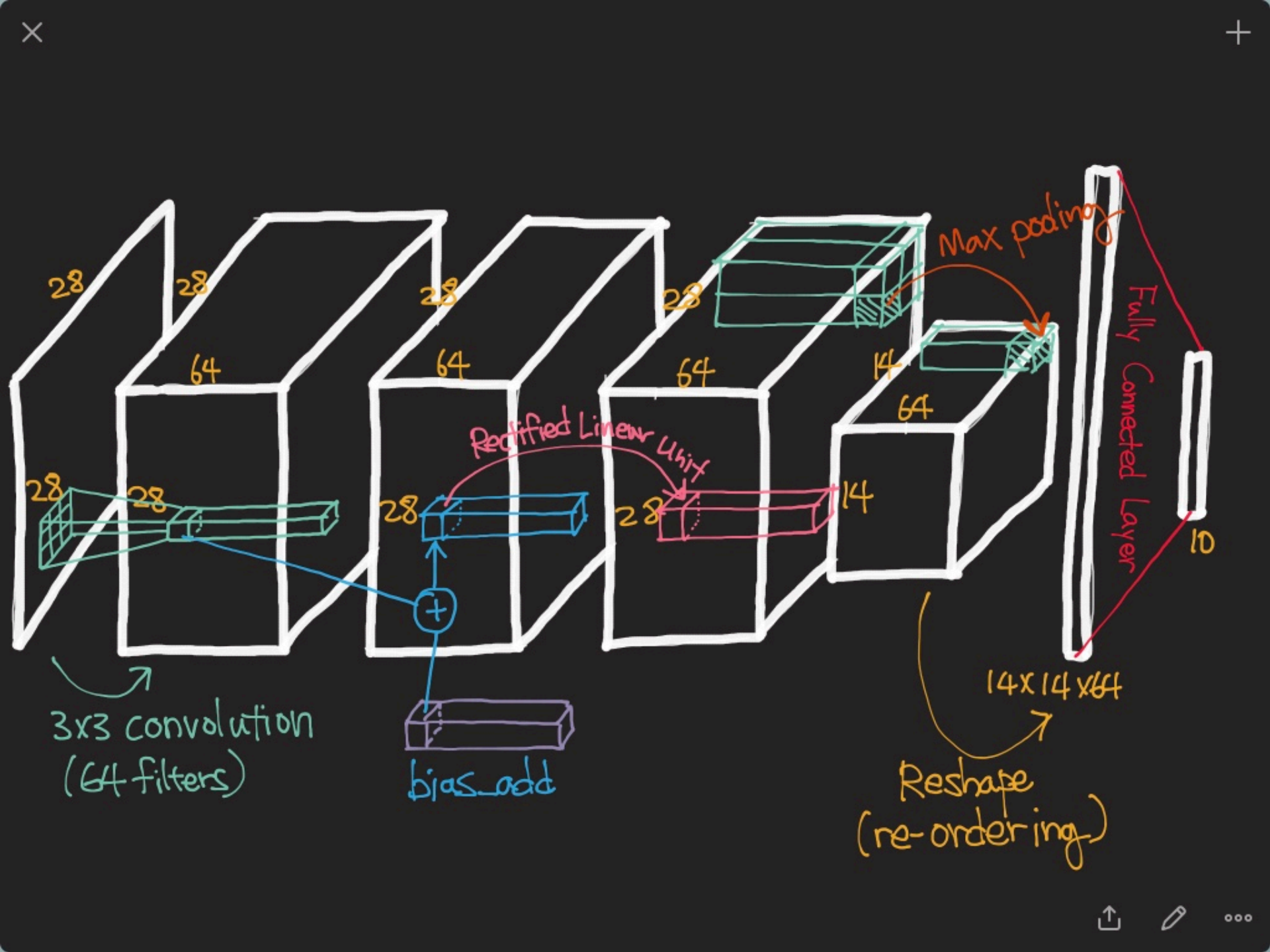

28
28
64
28
28
64
28
28
64
14
14
64
28
28
28
28
14
10
14x14x64
3x3 convolution
(64 filters)

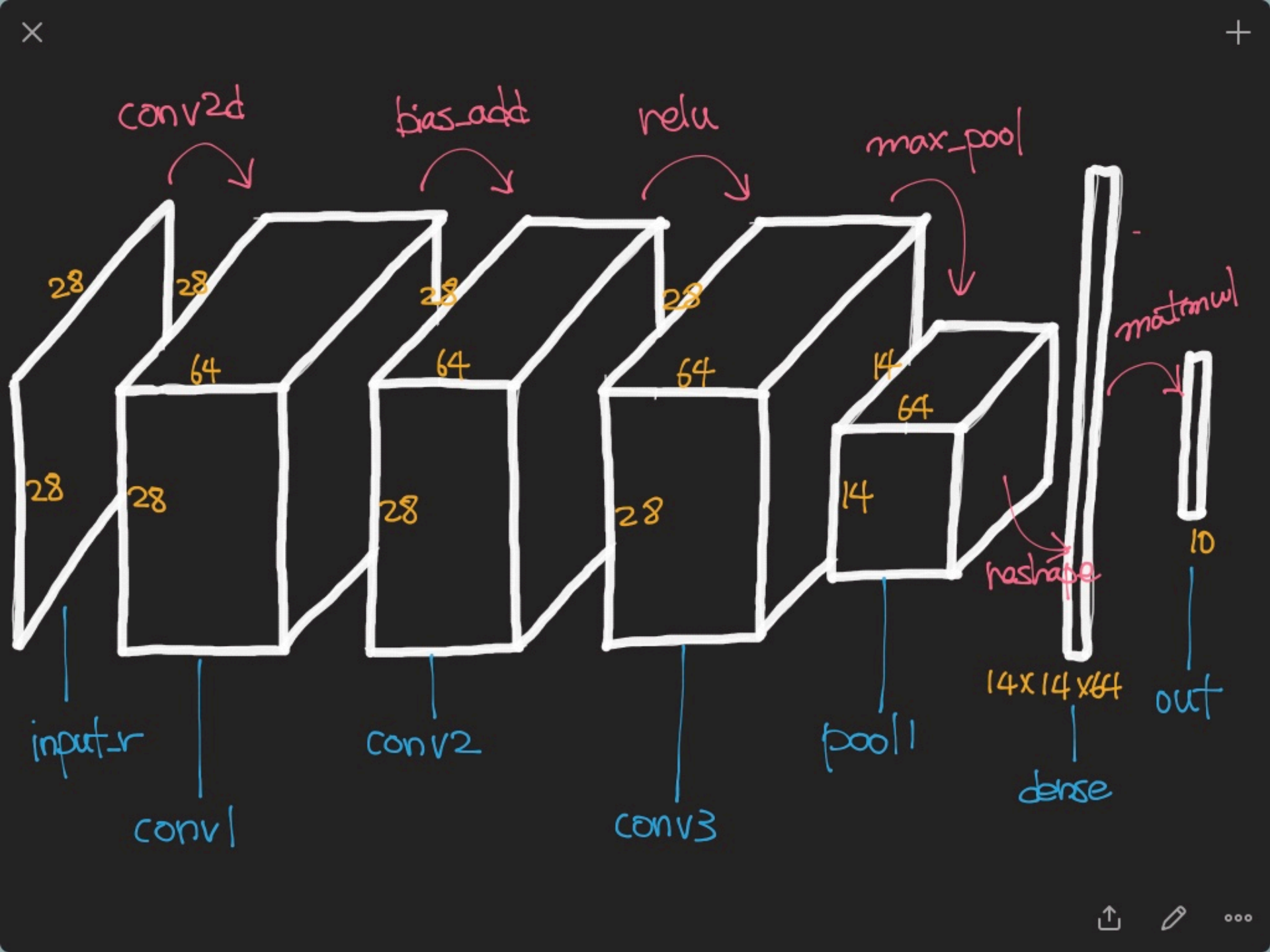

28
28
28
28
64
64
64
14
64
28
28
28
28
14
10
14X14X64
3x3 convolution
(64 filters)
bias_add

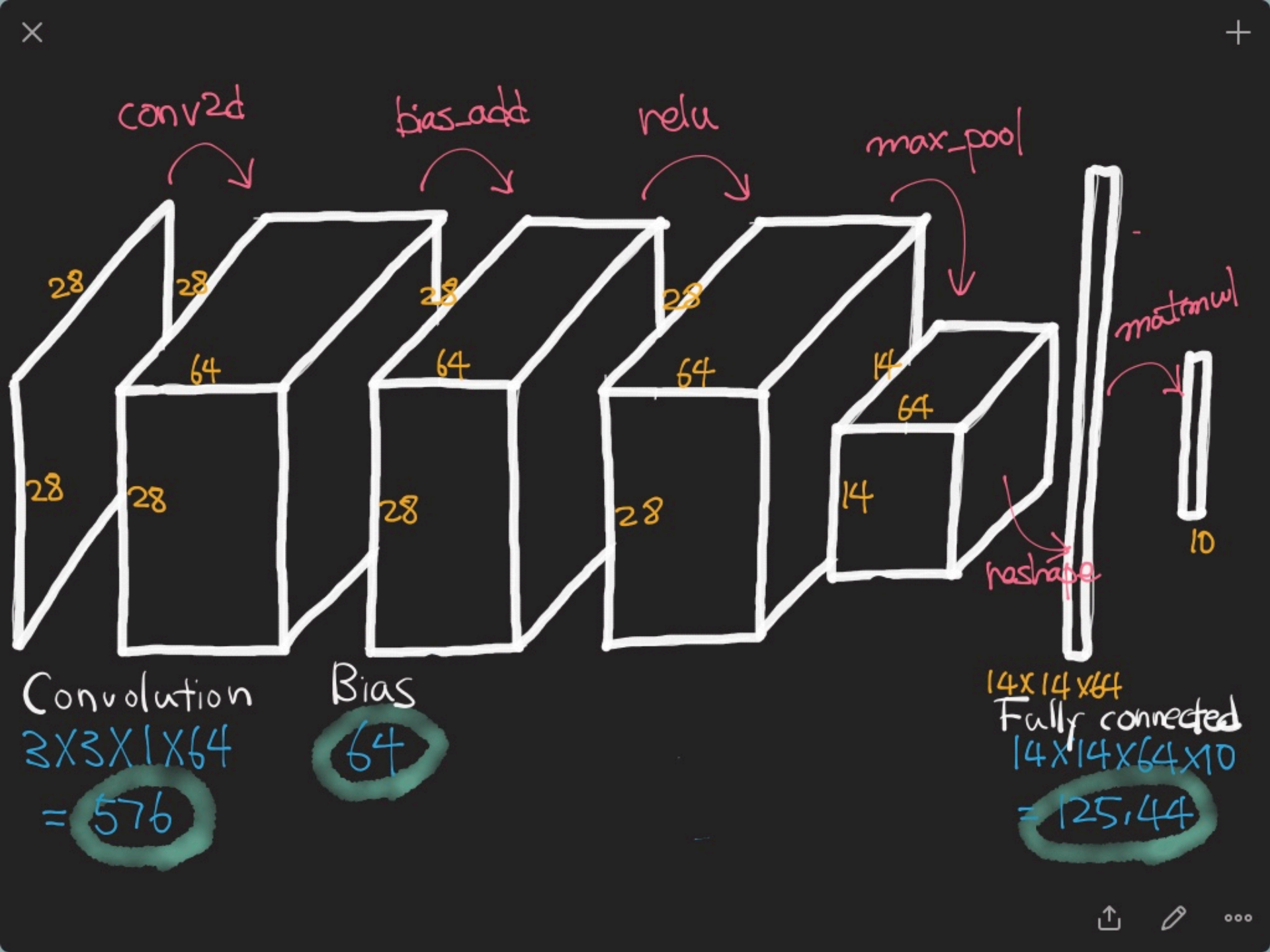

28
28
28
28
64
64
64
14
64
28
28
28
28
14
Rectified Linear Unit
10
14x14x64
3x3 convolution
(64 filters)
bias_add


28
28
64
28
28
64
28
28
64
14
14
64
Max pooling
Rectified Linear Unit
28
28
28
28
14
10
3x3 convolution
(64 filters)
bias_add
14x14x64


28
28
64
28
28
64
28
28
64
14
14
64
Max pooling
Rectified Linear Unit
28
28
28
28
14
10
3x3 convolution
(64 filters)
bias_add
14x14x64
Reshape
(re-ordering)


28
28
64
28
28
64
28
28
64
14
14
64
Max pooling
Fully Connected Layer
10
Rectified Linear Unit
28
28
28
28
14
14x14x64
3x3 convolution
(64 filters)
bias_add
Reshape
(re-ordering)


conv2d
bias_add
relu
max_pool
matmul
28
28
64
28
28
64
28
28
64
14
14
64
10
reshape
input_r
conv1
conv2
conv3
pool1
14x14x64
dense
out


conv2d
bias_add
relu
max_pool
matmul
28
28
64
28
28
28
28
64
28
28
64
14
14
64
14
10
reshape
Convolution
3X3X1X64
= 576
Bias
64
14X14X64
Fully connected
14X14X64X10
= 125,44

Implementing simple CNNs

SIMPLE CONVOLUTIONAL NEURAL NETWORK

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
%matplotlib inline
print ("PACKAGES LOADED")
```

```
PACKAGES LOADED
```

LOAD MNIST

```
mnist = input_data.read_data_sets('data/', one_hot=True)
trainimg   = mnist.train.images
trainlabel = mnist.train.labels
testimg    = mnist.test.images
testlabel  = mnist.test.labels
print ("MNIST ready")
```

```
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
MNIST ready
```

SELECT DEVICE TO BE USED

```
device_type = "/cpu:0"
```

DEFINE CNN

```
with tf.device('/cpu:0'): # <= This is optional
    n_input  = 784
    n_output = 10
    weights  = {
        'wc1': tf.Variable(tf.random_normal([3, 3, 1, 64], stddev=0.1)),
        'wd1': tf.Variable(tf.random_normal([14*14*64, n_output], stddev=0.1))
    }
    biases   = {
        'bc1': tf.Variable(tf.random_normal([64], stddev=0.1)),
        'bd1': tf.Variable(tf.random_normal([n_output], stddev=0.1))
    }
    def conv_simple(_input, _w, _b):
        # Reshape input
        _input_r = tf.reshape(_input, shape=[-1, 28, 28, 1])
        # Convolution
        _conv1 = tf.nn.conv2d(_input_r, _w['wc1'], strides=[1, 1, 1, 1], padding='SAME')
        # Add-bias
        _conv2 = tf.nn.bias_add(_conv1, _b['bc1'])
        # Pass ReLu
        _conv3 = tf.nn.relu(_conv2)
        # Max-pooling
        _pool  = tf.nn.max_pool(_conv3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        # Vectorize
        _dense = tf.reshape(_pool, [-1, _w['wd1'].get_shape().as_list()[0]])
        # Fully-connected layer
        _out = tf.add(tf.matmul(_dense, _w['wd1']), _b['bd1'])
        # Return everything
        out = {
            'input_r': _input_r, 'conv1': _conv1, 'conv2': _conv2, 'conv3': _conv3
            , 'pool': _pool, 'dense': _dense, 'out': _out
        }
        return out
print ("CNN ready")
```

```
CNN ready
```

DEFINE COMPUTATIONAL GRAPH

```
# tf Graph input
x = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_output])
# Parameters
learning_rate   = 0.001
training_epochs = 5
batch_size      = 100
display_step    = 1
# Functions!
with tf.device('/cpu:0'): # <= This is optional
    _pred = conv_simple(x, weights, biases)['out']
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(_pred, y))
    optm = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
    _corr = tf.equal(tf.argmax(_pred,1), tf.argmax(y,1)) # Count corrects
    accr = tf.reduce_mean(tf.cast(_corr, tf.float32)) # Accuracy
    init = tf.initialize_all_variables()
# Saver
save_step = 1;
savedir = "nets/"
saver = tf.train.Saver(max_to_keep=3)
print ("Network Ready to Go!")
```

```
Network Ready to Go!
```

OPTIMIZE

DO TRAIN OR NOT

```
do_train = 1
sess = tf.Session()
sess.run(init)
```

```
if do_train == 1:
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optm, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batch_ys})/total_batch

        # Display logs per epoch step
        if epoch % display_step == 0:
            print ("Epoch: %03d/%03d cost: %.9f" % (epoch, training_epochs, avg_cost))
            train_acc = sess.run(accr, feed_dict={x: batch_xs, y: batch_ys})
            print (" Training accuracy: %.3f" % (train_acc))
            test_acc = sess.run(accr, feed_dict={x: testimg, y: testlabel})
            print (" Test accuracy: %.3f" % (test_acc))

        # Save Net
        if epoch % save_step == 0:
            saver.save(sess, "nets/cnn_mnist_simple.ckpt-" + str(epoch))

    print ("Optimization Finished.")
```

```python
if do_train == 1:
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optm, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict

        # Display logs per epoch step
        if epoch % display_step == 0:
            print ("Epoch: %03d/%03d cost: %.9f"
            train_acc = sess.run(accr, feed_dict
            print (" Training accuracy: %.3f" %
            test_acc = sess.run(accr, feed_dict=
            print (" Test accuracy: %.3f" % (tes

        # Save Net
        if epoch % save_step == 0:
            saver.save(sess, "nets/cnn_mnist_sim

    print ("Optimization Finished.")
```

```
Epoch: 000/005 cost: 0.362760447
 Training accuracy: 0.960
 Test accuracy: 0.951
Epoch: 001/005 cost: 0.117072306
 Training accuracy: 0.980
 Test accuracy: 0.974
Epoch: 002/005 cost: 0.076811765
 Training accuracy: 0.960
 Test accuracy: 0.978
Epoch: 003/005 cost: 0.059349174
 Training accuracy: 0.980
 Test accuracy: 0.981
Epoch: 004/005 cost: 0.049465416
 Training accuracy: 0.980
 Test accuracy: 0.983
Optimization Finished.
```

RESTORE NETWORK

```
if do_train == 0:
    epoch = 4
    saver.restore(sess, "nets/cnn_mnist_simple.ckpt-" + str(epoch))
```

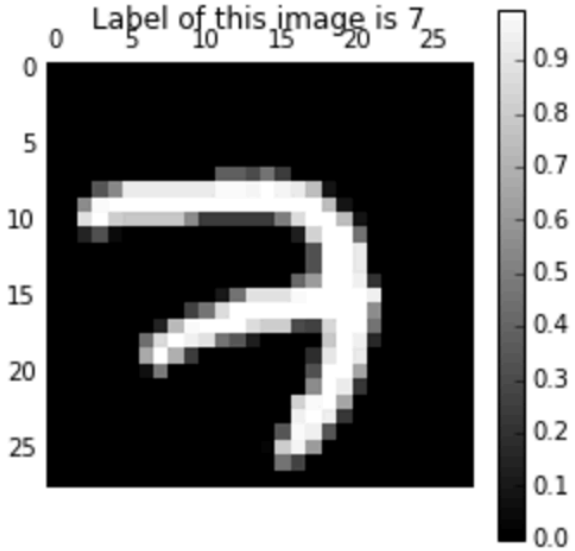
LET'S SEE HOW CNN WORKS

```
with tf.device('/cpu:0'):
    conv_out = conv_simple(x, weights, biases)

input_r = sess.run(conv_out['input_r'], feed_dict={x: trainimg[0:1, :]})
conv1   = sess.run(conv_out['conv1'], feed_dict={x: trainimg[0:1, :]})
conv2   = sess.run(conv_out['conv2'], feed_dict={x: trainimg[0:1, :]})
conv3   = sess.run(conv_out['conv3'], feed_dict={x: trainimg[0:1, :]})
pool    = sess.run(conv_out['pool'], feed_dict={x: trainimg[0:1, :]})
dense   = sess.run(conv_out['dense'], feed_dict={x: trainimg[0:1, :]})
out     = sess.run(conv_out['out'], feed_dict={x: trainimg[0:1, :]})
```

INPUT

```
print ("Size of 'input_r' is %s" % (input_r.shape,))
label = np.argmax(trainlabel[0, :])
print ("Label is %d" % (label))

# Plot !
plt.matshow(input_r[0, :, :, 0], cmap=plt.get_cmap('gray'))
plt.title("Label of this image is " + str(label) + "")
plt.colorbar()
plt.show()
```

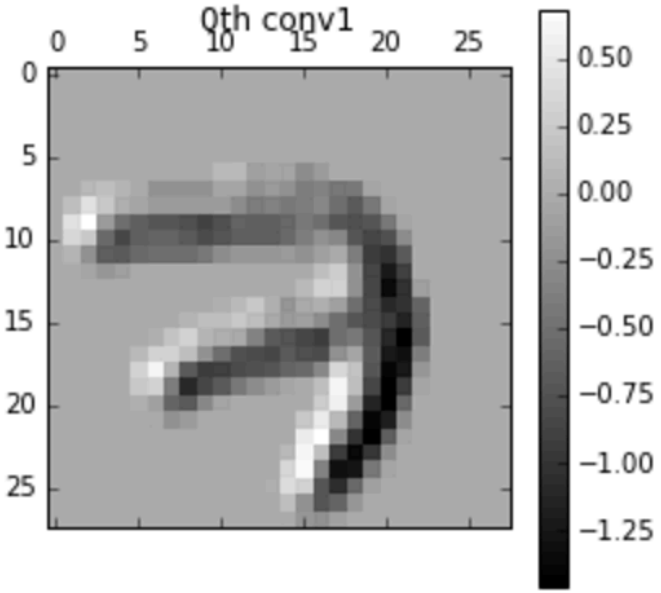
```
Size of 'input_r' is (1, 28, 28, 1)
Label is 7
```



CONV1 (CONVOLUTION)

```
print ("Size of 'conv1' is %s" % (conv1.shape,))

# PLOT
for i in range(64):
    plt.matshow(conv1[0, :, :, i], cmap=plt.get_cmap('gray'))
    plt.title(str(i) + "th conv1")
    plt.colorbar()
    plt.show()
```

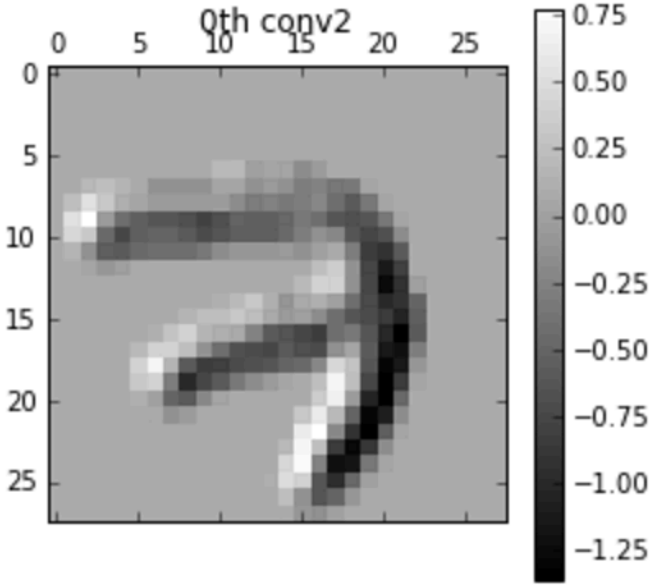
```
Size of 'conv1' is (1, 28, 28, 64)
```



CONV2 (+BIAS)

```
print ("Size of 'conv2' is %s" % (conv2.shape,))

# PLOT
for i in range(64):
    plt.matshow(conv2[0, :, :, i], cmap=plt.get_cmap('gray'))
    plt.title(str(i) + "th conv2")
    plt.colorbar()
    plt.show()
```

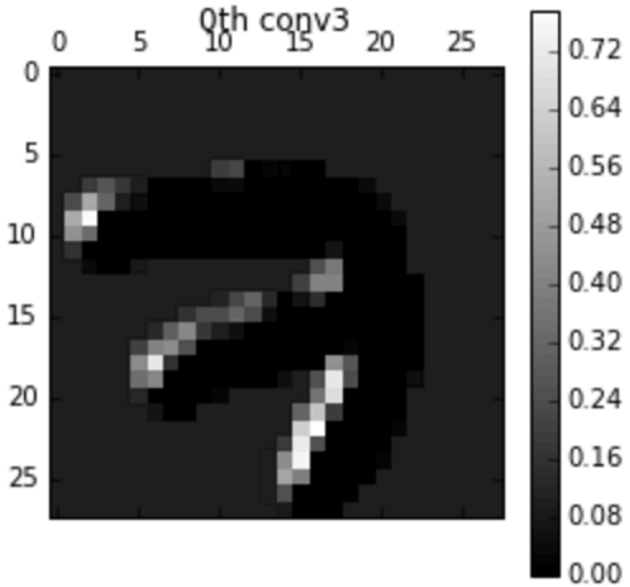
```
Size of 'conv2' is (1, 28, 28, 64)
```



CONV3 (RELU)

```
print ("Size of 'conv3' is %s" % (conv3.shape,))

# Plot !
for i in range(64):
    plt.matshow(conv3[0, :, :, i], cmap=plt.get_cmap('gray'))
    plt.title(str(i) + "th conv3")
    plt.colorbar()
    plt.show()
```

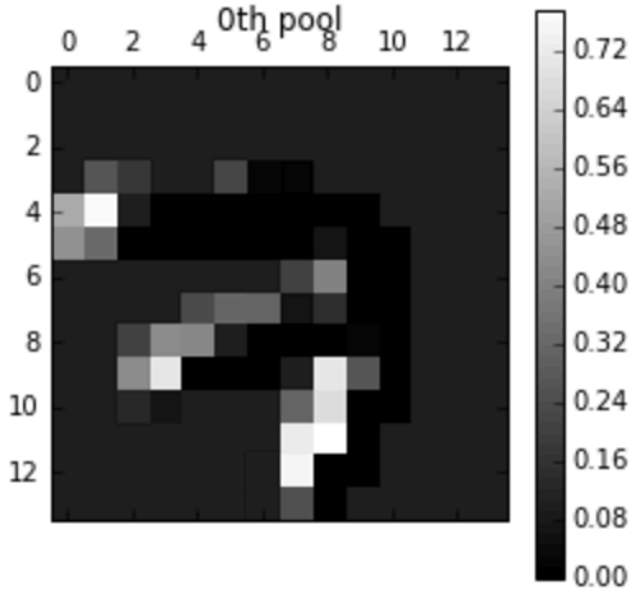
```
Size of 'conv3' is (1, 28, 28, 64)
```



POOL (MAX_POOL)

```
print ("Size of 'pool' is %s" % (pool.shape,))

# Plot !
for i in range(64):
    plt.matshow(pool[0, :, :, i], cmap=plt.get_cmap('gray'))
    plt.title(str(i) + "th pool")
    plt.colorbar()
    plt.show()
```

```
Size of 'pool' is (1, 14, 14, 64)
```



DENSE

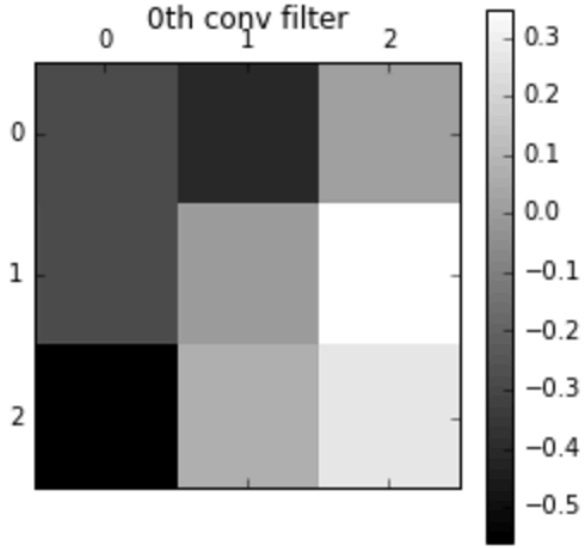
```
# Let's see 'dense'
print ("Size of 'dense' is %s" % (dense.shape,))
# Let's see 'out'
print ("Size of 'out' is %s" % (out.shape,))
```

```
Size of 'dense' is (1, 12544)
Size of 'out' is (1, 10)
```

CONVOLUTION FILTERS

```
# Let's see weight!
wc1 = sess.run(weights['wc1'])
print ("Size of 'wc1' is %s" % (wc1.shape,))

# Plot !
for i in range(64):
    plt.matshow(wc1[:, :, 0, i], cmap=plt.get_cmap('gray'))
    plt.title(str(i) + "th conv filter")
    plt.colorbar()
    plt.show()
```

```
Size of 'wc1' is (3, 3, 1, 64)
```



Implementing modern CNNs

- dropout + batch_norm

MODERN CONVOLUTIONAL NEURAL NETWORK

DROPOUT + BATCH NORMALIZATION

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
%matplotlib inline
print ("PACKAGES LOADED")
```

```
PACKAGES LOADED
```

LOAD MNIST

```
mnist = input_data.read_data_sets('data/', one_hot=True)
trainimg   = mnist.train.images
trainlabel = mnist.train.labels
testimg    = mnist.test.images
testlabel  = mnist.test.labels
```

```
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
```

SELECT DEVICE TO BE USED

```
device_type = "/gpu:0"
```

DEFINE CNN

```
n_input  = 784
n_output = 10
with tf.device(device_type):
    weights  = {
        'wc1': tf.Variable(tf.truncated_normal([3, 3, 1, 64], stddev=0.1)),
        'wc2': tf.Variable(tf.truncated_normal([3, 3, 64, 128], stddev=0.1)),
        'wd1': tf.Variable(tf.truncated_normal([7*7*128, 1024], stddev=0.1)),
        'wd2': tf.Variable(tf.truncated_normal([1024, n_output], stddev=0.1))
    }
    biases   = {
        'bc1': tf.Variable(tf.random_normal([64], stddev=0.1)),
        'bc2': tf.Variable(tf.random_normal([128], stddev=0.1)),
        'bd1': tf.Variable(tf.random_normal([1024], stddev=0.1)),
        'bd2': tf.Variable(tf.random_normal([n_output], stddev=0.1))
    }
    def conv_basic(_input, _w, _b, _keepratio):
        # INPUT
        _input_r = tf.reshape(_input, shape=[-1, 28, 28, 1])
        # CONV LAYER 1
        _conv1 = tf.nn.conv2d(_input_r, _w['wc1'], strides=[1, 1, 1, 1], padding='SAME')
        _mean, _var = tf.nn.moments(_conv1, [0, 1, 2])
        _conv1 = tf.nn.batch_normalization(_conv1, _mean, _var, 0, 1, 0.0001)
        _conv1 = tf.nn.relu(tf.nn.bias_add(_conv1, _b['bc1']))
        _pool1 = tf.nn.max_pool(_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        _pool_dr1 = tf.nn.dropout(_pool1, _keepratio)
        # CONV LAYER 2
        _conv2 = tf.nn.conv2d(_pool_dr1, _w['wc2'], strides=[1, 1, 1, 1], padding='SAME')
        _mean, _var = tf.nn.moments(_conv2, [0, 1, 2])
        _conv2 = tf.nn.batch_normalization(_conv2, _mean, _var, 0, 1, 0.0001)
        _conv2 = tf.nn.relu(tf.nn.bias_add(_conv2, _b['bc2']))
        _pool2 = tf.nn.max_pool(_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        _pool_dr2 = tf.nn.dropout(_pool2, _keepratio)
        # VECTORIZE
        _dense1 = tf.reshape(_pool_dr2, [-1, _w['wd1'].get_shape().as_list()[0]])
        # FULLY CONNECTED LAYER 1
        _fc1 = tf.nn.relu(tf.add(tf.matmul(_dense1, _w['wd1']), _b['bd1']))
        _fc_dr1 = tf.nn.dropout(_fc1, _keepratio)
        # FULLY CONNECTED LAYER 2
        _out = tf.add(tf.matmul(_fc_dr1, _w['wd2']), _b['bd2'])
        # RETURN
        out = { 'input_r': _input_r, 'conv1': _conv1, 'pool1': _pool1, 'pool1_dr1': _pool_dr1,
            'conv2': _conv2, 'pool2': _pool2, 'pool_dr2': _pool_dr2, 'dense1': _dense1,
            'fc1': _fc1, 'fc_dr1': _fc_dr1, 'out': _out
        }
        return out
print ("CNN READY")
```

```
weights  = {
    'wc1': tf.Variable(tf.truncated_normal([3, 3, 1, 64], stddev=0.1)),
    'wc2': tf.Variable(tf.truncated_normal([3, 3, 64, 128], stddev=0.1)),
    'wd1': tf.Variable(tf.truncated_normal([7*7*128, 1024], stddev=0.1)),
    'wd2': tf.Variable(tf.truncated_normal([1024, n_output], stddev=0.1))
}
biases   = {
    'bc1': tf.Variable(tf.random_normal([64], stddev=0.1)),
    'bc2': tf.Variable(tf.random_normal([128], stddev=0.1)),
    'bd1': tf.Variable(tf.random_normal([1024], stddev=0.1)),
    'bd2': tf.Variable(tf.random_normal([n_output], stddev=0.1))
}
```

```python
def conv_basic(_input, _w, _b, _keepratio):
    # INPUT
    _input_r = tf.reshape(_input, shape=[-1, 28, 28, 1])
    # CONV LAYER 1
    _conv1 = tf.nn.conv2d(_input_r, _w['wc1'], strides=[1, 1, 1, 1], padding='SAME')
    _mean, _var = tf.nn.moments(_conv1, [0, 1, 2])
    _conv1 = tf.nn.batch_normalization(_conv1, _mean, _var, 0, 1, 0.0001)
    _conv1 = tf.nn.relu(tf.nn.bias_add(_conv1, _b['bc1']))
    _pool1 = tf.nn.max_pool(_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    _pool_dr1 = tf.nn.dropout(_pool1, _keepratio)
    # CONV LAYER 2
    _conv2 = tf.nn.conv2d(_pool_dr1, _w['wc2'], strides=[1, 1, 1, 1], padding='SAME')
    _mean, _var = tf.nn.moments(_conv2, [0, 1, 2])
    _conv2 = tf.nn.batch_normalization(_conv2, _mean, _var, 0, 1, 0.0001)
    _conv2 = tf.nn.relu(tf.nn.bias_add(_conv2, _b['bc2']))
    _pool2 = tf.nn.max_pool(_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    _pool_dr2 = tf.nn.dropout(_pool2, _keepratio)
    # VECTORIZE
    _dense1 = tf.reshape(_pool_dr2, [-1, _w['wd1'].get_shape().as_list()[0]])
    # FULLY CONNECTED LAYER 1
    _fc1 = tf.nn.relu(tf.add(tf.matmul(_dense1, _w['wd1']), _b['bd1']))
    _fc_dr1 = tf.nn.dropout(_fc1, _keepratio)
    # FULLY CONNECTED LAYER 2
    _out = tf.add(tf.matmul(_fc_dr1, _w['wd2']), _b['bd2'])
    # RETURN
    out = { 'input_r': _input_r, 'conv1': _conv1, 'pool1': _pool1, 'pool1_dr1': _pool_dr1,
        'conv2': _conv2, 'pool2': _pool2, 'pool_dr2': _pool_dr2, 'dense1': _dense1,
        'fc1': _fc1, 'fc_dr1': _fc_dr1, 'out': _out
    }
    return out
```

tf.nn.dropout

```
tf.nn.dropout(x, keep_prob, noise_shape=None,
seed=None, name=None)
```

Computes dropout.

With probability `keep_prob`, outputs the input element scaled up by `1 / keep_prob`, otherwise outputs `0`. The scaling is so that the expected sum is unchanged.

By default, each element is kept or dropped independently. If `noise_shape` is specified, it must be broadcastable to the shape of `x`, and only dimensions with `noise_shape[i] == shape(x)[i]` will make independent decisions. For example, if `shape(x) = [k, l, m, n]` and `noise_shape = [k, 1, 1, n]`, each batch and channel component will be kept independently and each row and column will be kept or not kept together.

Args:

- `x`: A tensor.
- `keep_prob`: A scalar `Tensor` with the same type as x. The probability that each element is kept.
- `noise_shape`: A 1-D `Tensor` of type `int32`, representing the shape for randomly generated keep/drop flags.
- `seed`: A Python integer. Used to create random seeds. See `set_random_seed` for behavior.
- `name`: A name for this operation (optional).

Returns:

A Tensor of the same shape of `x`.

Raises:

- `ValueError`: If `keep_prob` is not in `(0, 1]`.

tf.nn.moments

`tf.nn.moments(x, axes, shift=None, name=None, keep_dims=False)`

Calculate the mean and variance of `x`.

The mean and variance are calculated by aggregating the contents of `x` across `axes`. If `x` is 1-D and `axes = [0]` this is just the mean and variance of a vector.

When using these moments for batch normalization (see `tf.nn.batch_normalization`): * for so-called "global normalization", used with convolutional filters with shape `[batch, height, width, depth]`, pass `axes=[0, 1, 2]`. * for simple batch normalization pass `axes=[0]` (batch only).

Args:

- `x`: A `Tensor`.
- `axes`: array of ints. Axes along which to compute mean and variance.
- `shift`: A `Tensor` containing the value by which to shift the data for numerical stability, or `None` if no shift is to be performed. A shift close to the true mean provides the most numerically stable results.
- `name`: Name used to scope the operations that compute the moments.
- `keep_dims`: produce moments with the same dimensionality as the input.

Returns:

Two `Tensor` objects: `mean` and `variance`.

tf.nn.batch_normalization

```
tf.nn.batch_normalization(x, mean, variance, offset,
scale, variance_epsilon, name=None)
```

Batch normalization.

As described in http://arxiv.org/abs/1502.03167. Normalizes a tensor by `mean` and `variance`, and applies (optionally) a `scale` γ to it, as well as an `offset` $beta$:

$$\frac{\gamma(x-\mu)}{\sigma} + beta$$

`mean`, `variance`, `offset` and `scale` are all expected to be of one of two shapes: * In all generality, they can have the same number of dimensions as the input `x`, with identical sizes as `x` for the dimensions that are not normalized over (the 'depth' dimension(s)), and dimension 1 for the others which are being normalized over. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=True)` during training, or running averages thereof during inference. * In the common case where the 'depth' dimension is the last dimension in the input tensor `x`, they may be one dimensional tensors of the same size as the 'depth' dimension. This is the case for example for the common `[batch, depth]` layout of fully-connected layers, and `[batch, height, width, depth]` for convolutions. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=False)` during training, or running averages thereof during inference.

DEFINE COMPUTATIONAL GRAPH

```
# PLACEHOLDERS
x = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_output])
keepratio = tf.placeholder(tf.float32)

# FUNCTIONS
with tf.device(device_type):
    _pred = conv_basic(x, weights, biases, keepratio)['out']
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(_pred, y))
    optm = tf.train.AdamOptimizer(learning_rate=0.001).minimize(cost)
    _corr = tf.equal(tf.argmax(_pred,1), tf.argmax(y,1))
    accr = tf.reduce_mean(tf.cast(_corr, tf.float32))
    init = tf.initialize_all_variables()

# SAVER
save_step = 1
saver = tf.train.Saver(max_to_keep=3)
print ("GRAPH READY")
```

```
GRAPH READY
```

OPTIMIZE

DO TRAIN OR NOT

```
do_train = 1
sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True))
sess.run(init)
```

```
training_epochs = 10
batch_size      = 100
display_step    = 1
if do_train == 1:
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optm, feed_dict={x: batch_xs, y: batch_ys, keepratio:0.7})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batch_ys, keepratio:1.})/total_batch

        # Display logs per epoch step
        if epoch % display_step == 0:
            print ("Epoch: %03d/%03d cost: %.9f" % (epoch, training_epochs, avg_cost))
            train_acc = sess.run(accr, feed_dict={x: batch_xs, y: batch_ys, keepratio:1.})
            print (" Training accuracy: %.3f" % (train_acc))
            test_acc = sess.run(accr, feed_dict={x: testimg, y: testlabel, keepratio:1.})
            print (" Test accuracy: %.3f" % (test_acc))

        # Save Net
        if epoch % save_step == 0:
            saver.save(sess, "nets/cnn_mnist_basic.ckpt-" + str(epoch))

    print ("OPTIMIZATION FINISHED")
```

```
training_epochs = 10
batch_size      = 100
display_step    = 1
if do_train == 1:
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_s
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(ba
            # Fit training using batch data
            sess.run(optm, feed_dict={x: batch_xs, y: batc
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch

        # Display logs per epoch step
        if epoch % display_step == 0:
            print ("Epoch: %03d/%03d cost: %.9f" % (epoch,
            train_acc = sess.run(accr, feed_dict={x: batch
            print (" Training accuracy: %.3f" % (train_acc
            test_acc = sess.run(accr, feed_dict={x: testim
            print (" Test accuracy: %.3f" % (test_acc))

        # Save Net
        if epoch % save_step == 0:
            saver.save(sess, "nets/cnn_mnist_basic.ckpt-"

    print ("OPTIMIZATION FINISHED")
```

```
Epoch: 000/010 cost: 0.646904730
 Training accuracy: 0.980
 Test accuracy: 0.963
Epoch: 001/010 cost: 0.093934917
 Training accuracy: 0.970
 Test accuracy: 0.976
Epoch: 002/010 cost: 0.065349482
 Training accuracy: 0.990
 Test accuracy: 0.984
Epoch: 003/010 cost: 0.051335570
 Training accuracy: 0.990
 Test accuracy: 0.983
Epoch: 004/010 cost: 0.041595699
 Training accuracy: 0.960
 Test accuracy: 0.988
Epoch: 005/010 cost: 0.035545130
 Training accuracy: 0.990
 Test accuracy: 0.989
Epoch: 006/010 cost: 0.031783653
 Training accuracy: 0.970
 Test accuracy: 0.989
Epoch: 007/010 cost: 0.028224952
 Training accuracy: 1.000
 Test accuracy: 0.991
Epoch: 008/010 cost: 0.024041305
 Training accuracy: 0.990
 Test accuracy: 0.990
Epoch: 009/010 cost: 0.023118890
 Training accuracy: 0.990
 Test accuracy: 0.991
OPTIMIZATION FINISHED
```

RESTORE ¶

```
if do_train == 0:
    epoch = training_epochs-1
    saver.restore(sess, "nets/cnn_mnist_basic.ckpt-" + str(epoch))
```

COMPUTE TEST ACCURACY

```
test_acc = sess.run(accr, feed_dict={x: testimg, y: testlabel, keepratio:1.})
print (" TEST ACCURACY: %.3f" % (test_acc))
```

```
 TEST ACCURACY: 0.991
```