Sparse Distributed Memory Framework Documentation

Alexandre Linhares, Marcelo Salhab Brogliato

Aug 18, 2018

Contents

Welcome! This is the documentation for Sparse Distributed Memory Framework.

CHAPTER 1

Parts of the documentation

The documentation is organized in two parts: C API and Python Library.

It is suggested to use the Python Library. It is a facade which simplifies the creation and manipulation of SDM instances. It also works in the Jupyter Notebook.

The C API is the low level module which may be extended to add new operations to the SDM.

1.1 SDM Library

1.1.1 `Bitstring`

1.1.2 `AddressSpace`

1.1.3 `Counter`

1.1.4 `SDM`

1.2 C API

1.2.1 Bitstring's functions

The following functions are used to create, destroy, and manipulate bitstrings. Many of them have a parameter *len*, which is the number of bytes of the bitstring. It was a design choice which saves computing it everytime.

typedef uint64_t **`bitstring_t`**

void **`bs_init_bitcount_table`**()

Initialize a 64kb in RAM which is used to improve performance when calculating the distance between two bitstrings.

*bitstring_t** **bs_alloc** (const unsigned int *len*)
Allocate memory for a bitstring with *len* bytes.

void **bs_free** (*bitstring_t *bs*)
Free the memory of a bitstring.

void **bs_copy** (*bitstring_t *dst*, const *bitstring_t *src*, unsigned int *len*)
Copy one bitstring into another.

void **bs_init_zeros** (*bitstring_t *bs*, unsigned int *len*, unsigned int *bits_remaining*)
Initialize a bitstring with all bits equal to zero. The bitstring's memory must have already been allocated.

void **bs_init_ones** (*bitstring_t *bs*, unsigned int *len*, unsigned int *bits_remaining*)
Initialize a bitstring with all bits equal to one. The bitstring's memory must have already been allocated.

void **bs_init_random** (*bitstring_t *bs*, unsigned int *len*, unsigned int *bits_remaining*)
Initialize a bitstring with random bits. Each bit is sampled from Bernoulli trial with p=0.5. The bitstring's memory must have already been allocated.

void **bs_init_hex** (*bitstring_t *bs*, unsigned int *len*, char *hex*)
Initialize a bitstring with random bits. Each bit is sampled from Bernoulli trial with p=0.5. The bitstring's memory must have already been allocated.

void **bs_init_b64** (*bitstring_t *bs*, char *b64*)
Initialize a bitstring from a base64 string. The bitstring's memory must have already been allocated.

void **bs_to_hex** (char *buf*, *bitstring_t *bs*, unsigned int *len*)
Initialize a bitstring from a hexadecimal string. The bitstring's memory must have already been allocated.

void **bs_to_b64** (char *buf*, *bitstring_t *bs*, unsigned int *len*)
Generate the base64 string representation of the bitstring.

int **bs_distance** (const *bitstring_t *bs1*, const *bitstring_t *bs2*, const unsigned int *len*)
Calculate the hamming distance between two bitstrings.

unsigned int **bs_get_bit** (*bitstring_t *bs*, unsigned int *bit*)
Return a specific bit from a bitstring.

void **bs_set_bit** (*bitstring_t *bs*, unsigned int *bit*, unsigned int *value*)
Change the value of a specific bit from a bitstring.

void **bs_flip_bit** (*bitstring_t *bs*, unsigned int *bit*)
Flip a specific bit from a bitstring.

int **bs_flip_random_bits** (*bitstring_t *bs*, unsigned int *bits*, unsigned int *flips*)
Randomly choose *flips* bits of the bitstring. It is used to generate a random bitstring with a given distance from another bitstring.

void **bs_xor** (*bitstring_t *bs1*, const *bitstring_t *bs2*, const unsigned int *len*)
Calculate the XOR bitwise operation between two bitstrings. The result is stored in *bs1*.

void **bs_and** (*bitstring_t *bs1*, const *bitstring_t *bs2*, const unsigned int *len*)
Calculate the AND bitwise operation between two bitstrings. The result is stored in *bs1*.

void **bs_or** (*bitstring_t *bs1*, const *bitstring_t *bs2*, const unsigned int *len*)
Calculate the OR bitwise operation between two bitstrings. The result is stored in *bs1*.

void **bs_average** (*bitstring_t *bs1*, const *bitstring_t *bs2*, const unsigned int *len*)
Calculate average between the bitstrings. The result is stored in *bs1*.

1.2.2 Address Space's functions

struct **address_space_s** ()

unsigned int **bits**
SDM dimension.

unsigned int **sample**
Number of hard-locations.

bitstring_t ****addresses**
This approach allocates a continuous chunk of memory for all bitstring addresses. The *addresses* allows the use of array notation: addresses[0], addresses[1], ...

Let *a* be *addresses*. Then:

```
          a[0]   a[1]   a[2]   a[3]   a[4]
          |      |      |      |      |
          v      v      v      v      v
bs_data = xxxxxx|xxxxxx|xxxxxx|xxxxxx|xxxxxx
```

unsigned int bs_len;

unsigned int bs_bits_remaining;

bitstring_t *bs_data;

int **as_init** (struct *address_space_s* *this, unsigned int *bits*, unsigned int *sample*)
Testing...

int **as_init_random** (struct *address_space_s* *this, unsigned int *bits*, unsigned int *sample*)
Testing again..

int **as_init_from_b64_file** (struct *address_space_s* *this, char *filename*)

int **as_free** (struct *address_space_s* *this)

int **as_save_b64_file** (const struct *address_space_s* *this, char *filename*)

int **as_scan_linear** (const struct *address_space_s* *this, const *bitstring_t* *bs, unsigned int *radius*, uint8_t *buf*)

int **as_scan_thread** (const struct *address_space_s* *this, const *bitstring_t* *bs, unsigned int *radius*, uint8_t *buf*, unsigned int *thread_count*)

void **as_print_summary** (struct *address_space_s* *this)

void **as_print_addresses_b64** (struct *address_space_s* *this)

void **as_print_addresses_hex** (struct *address_space_s* *this)

1.2.3 OpenCL Scanner

int **as_scanner_opencl_init** (struct opencl_scanner_s *this, struct *address_space_s* *as, char *opencl_source*)

void **as_scanner_opencl_free** (struct opencl_scanner_s *this)

int **as_scan_opencl** (struct opencl_scanner_s *this, *bitstring_t* *bs, unsigned int *radius*, uint8_t *result*)

1.2.4 Counter's functions

typedef int **counter_t**

struct **counter_s**

unsigned int **bits**

unsigned int **sample**

int **fd**

char ***filename**

counter_t ****counter**

counter_t ***data**

int **counter_init** (struct *counter_s* **this*, unsigned int *bits*, unsigned int *sample*)

int **counter_init_file** (char **filename*, struct *counter_s* **this*)

void **counter_free** (struct *counter_s* **this*)

void **counter_print_summary** (struct *counter_s* **this*)

void **counter_print** (struct *counter_s* **this*, unsigned int *index*)

int **counter_add_bitstring** (struct *counter_s* **this*, unsigned int *index*, *bitstring_t* **bs*)

int **counter_add_counter** (struct *counter_s* **c1*, unsigned int *idx1*, struct *counter_s* **c2*, unsigned int *idx2*)

int **counter_to_bitstring** (struct *counter_s* **this*, unsigned int *index*, *bitstring_t* **bs*)

int **counter_create_file** (char **filename*, unsigned int *bits*, unsigned int *sample*)

1.2.5 SDM's functions

struct **sdm_s**

unsigned int **bits**

unsinged int **sample**

unsinged int **scanner_type**

SDM_SCANNER_LINEAR

SDM_SCANNER_THREAD

SDM_SCANNER_OPENCL

struct opencl_scanner_s ***opencl_opts**

unsinged int **thread_count**

struct *address_space_s* ***address_space**

struct *counter_s* ***counter**

int **sdm_init_linear** (struct *sdm_s* **sdm*, struct *address_space_s* **address_space*, struct *counter_s* **counter*)

int **sdm_init_thread**(struct *sdm_s* *sdm, struct *address_space_s* *address_space, struct *counter_s* *counter, unsigned int *thread_count*)

int **sdm_init_opencl**(struct *sdm_s* *sdm, struct *address_space_s* *address_space, struct *counter_s* *counter, char *opencl_source*)

void **sdm_free**(struct *sdm_s* *sdm)

int **sdm_write**(struct *sdm_s* *sdm, *bitstring_t* *addr, unsigned int *radius*, *bitstring_t* *datum*)

int **sdm_read**(struct *sdm_s* *sdm, *bitstring_t* *addr, unsigned int *radius*, *bitstring_t* *output*)

int **sdm_iter_read**(struct *sdm_s* *sdm, *bitstring_t* *addr, unsigned int *radius*, unsigned int *max_iter*, *bitstring_t* *output*)

1.3 Examples

1.3.1 Distance between bitstrings

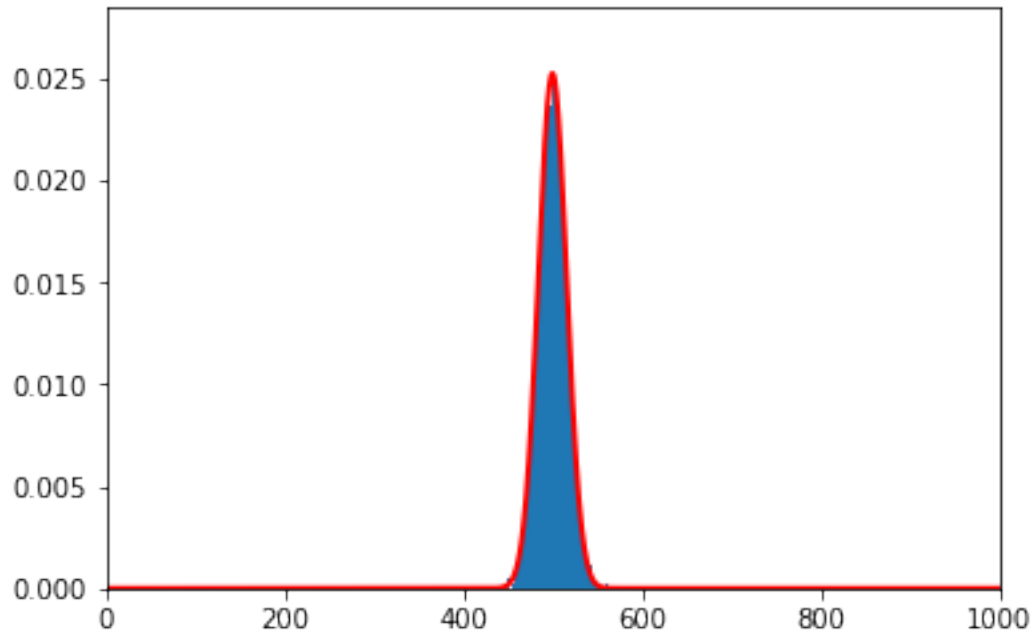
Here we will draw the histogram of the distance between two random bitstrings.

```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        import numpy as np
        import matplotlib.mlab as mlab
        #%matplotlib inline
```

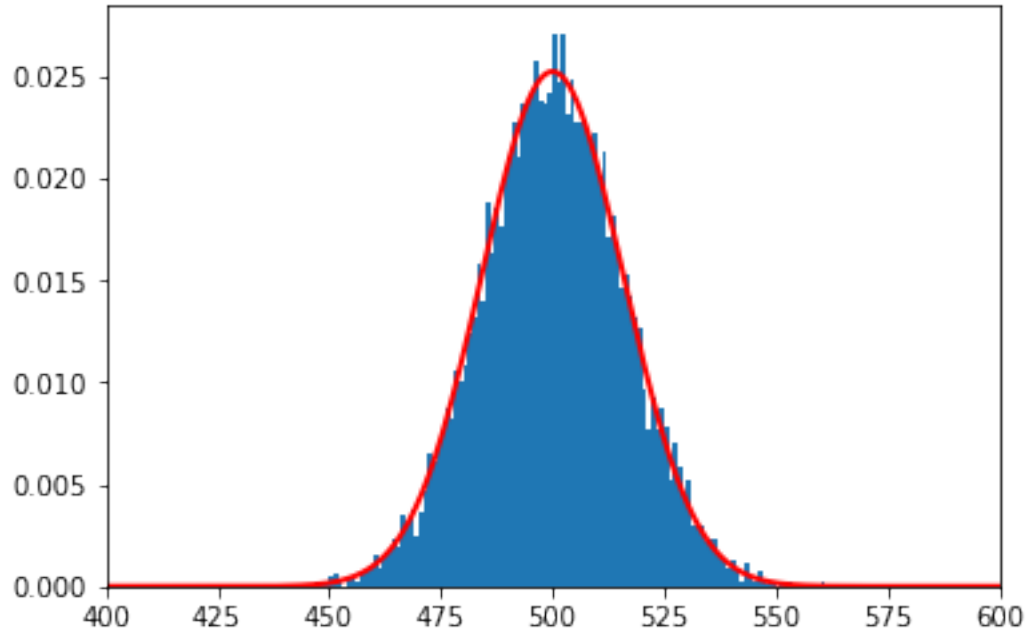
```
In [2]: distances = []
        for i in xrange(10000):
            b1 = sdmlib.Bitstring.init_random(1000)
            b2 = sdmlib.Bitstring.init_random(1000)
            distances.append(b1.distance_to(b2))
```

```
In [3]: mu = 500
        sigma = (1000**(0.5))/2.0
        x = np.linspace(0, 1000, 1000)
        y = mlab.normpdf(x, mu, sigma)
```

```
In [4]: plt.hist(distances, bins=range(1001), density=True)
        plt.plot(x, y, 'r', linewidth=2.0)
        plt.xlim(0, 1000)
        plt.show()
```




```
In [5]: plt.hist(distances, bins=range(1001), density=True)
        plt.plot(x, y, 'r', linewidth=2.0)
        plt.xlim(400, 600)
        plt.show()
```



1.3.2 Generates a new SDM

```
In [1]: import sdm as sdmlib
```

```
In [2]: bits = 1000
        sample = 1000000
```

```
        scanner_type = sdmlib.SDM_SCANNER_OPENCL
```

```
In [3]: # Generate an address space with 1,000,000 random 1,000-bit bitstrings.
        address_space = sdmlib.AddressSpace.init_random(bits, sample)

        # Generate 1,000,000 counters initialized with value zero in the RAM memory.
        counter = sdmlib.Counter.init_zero(bits, sample)

        # Create a file to store the 1,000,000 counters initialized with value zero.
        # You do not need to provide file extension, because it will generate two files
        # and automatically included the extension for you.
        #counter = sdmlib.Counter.create_file('sdm-10w', bits, sample)

        # Create an SDM with the generated address space and counter.
        # The scans will be performed using the OpenCL scanner.
        sdm = sdmlib.SDM(address_space, counter, 451, scanner_type)
```

```
In [4]: v = []
        for i in xrange(10):
            print i,
            # Generate a random 1,000-bit bitstring.
            b = sdmlib.Bitstring.init_random(1000)

            # Write the bitstring to the SDM.
            sdm.write(b, b)
            v.append(b)
        print ''
        print '{} bitstring wrote into memory.'.format(len(v))
```

```
0 1 2 3 4 5 6 7 8 9
10 bitstring wrote into memory.
```

```
In [5]: # Copy the bitstring from v[0].
        # We have to make a copy because the flip_random_bits function changes the bitstring itself.
        b = sdmlib.Bitstring.init_from_bitstring(v[0])
        b.flip_random_bits(400)

        # Read the bitstring from the SDM and checks the distance from the retrieved bitstring and v
        c = sdm.read(b)
        print 'Distance', c.distance_to(v[0])
```

```
Distance 134
```

```
In [6]: # Save the address space into the file 'sdm-10w.as'.
        # The recommended extension for an address space is '.as'.

        # Although we have used 10w as an indication of 10 writes to the memory, the
        # address space is not affected by the writes. It is just a reference to help us
        # remeber that this address space has been used together with the counters.
        #address_space.save('sdm-10w.as');
```

1.3.3 Kanerva's Figure 1.2 (page 25)

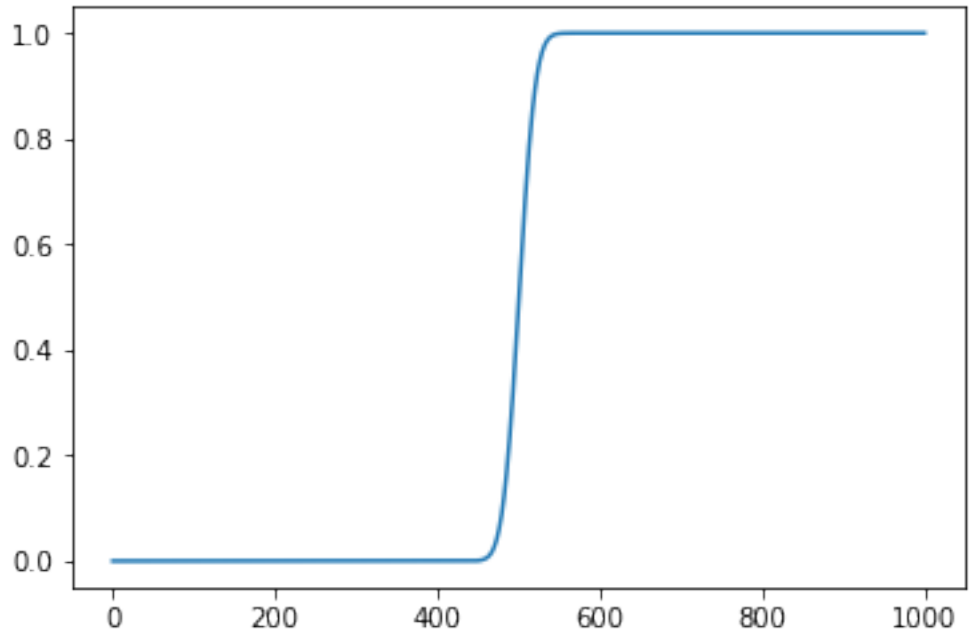
```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from IPython.display import clear_output
        %matplotlib inline
```

```
In [2]: bits = 1000
        sample = 1000000
```

```
        address_space = sdmlib.AddressSpace.init_random(bits, sample)
```

In [3]:
```
def calculate_probabilities():
    from math import factorial
    comb = lambda a, b: factorial(a)/factorial(b)/factorial(a-b)
    acc = [0]
    for i in xrange(1001):
        acc.append(acc[-1] + comb(1000, i))
    x = range(0, 1001)
    y = [acc[i]/float(2**1000) for i in xrange(1001)]
    return x, y
```

In [4]:
```
x, y = calculate_probabilities()
plt.plot(x, y);
```

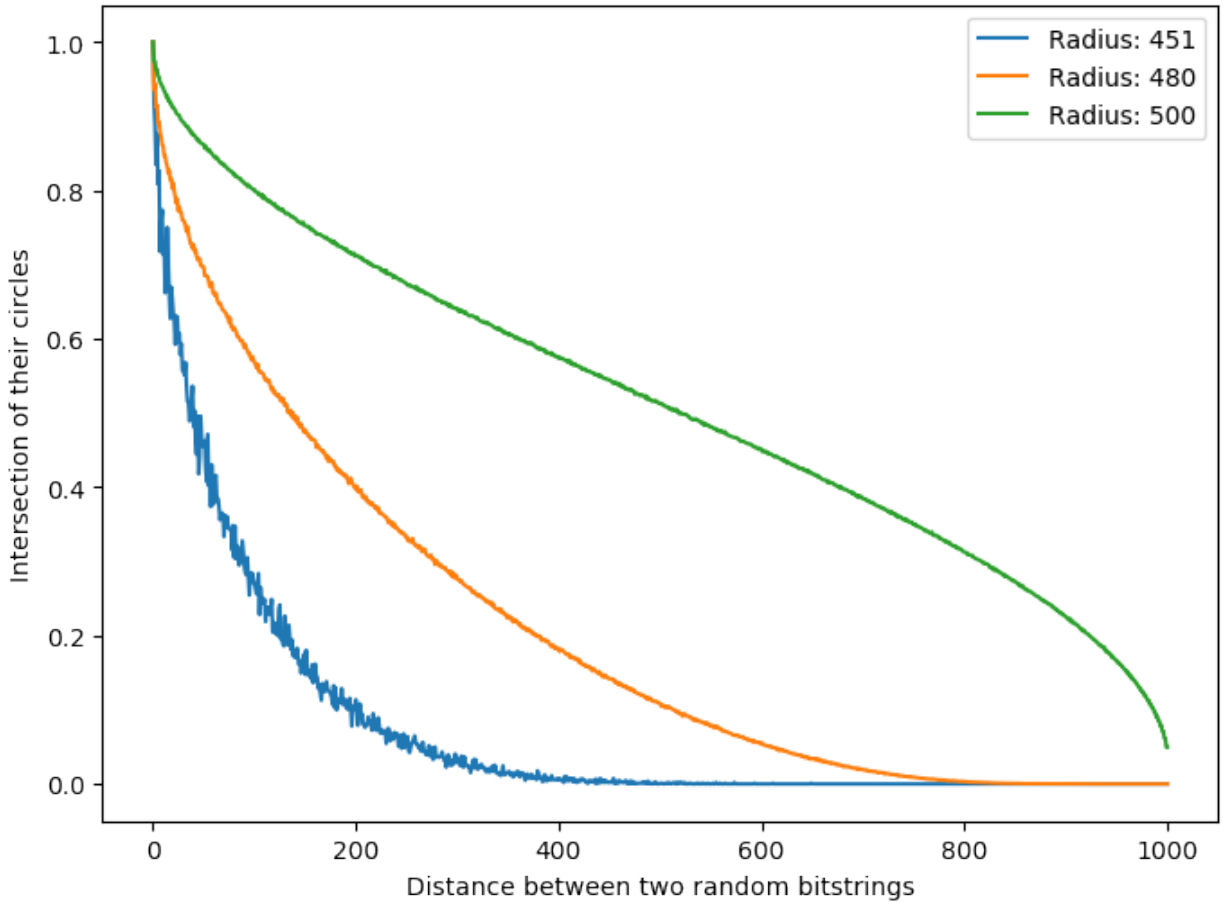


In [5]:
```
def run(radius):
    x = range(0, 1001)
    y = []
    for i, dist in enumerate(x):
        clear_output(wait=True)
        print 'Distance: {:4d} ({:.2f}%)'.format(dist, 100.*(i+1)/len(x))

        b1 = sdmlib.Bitstring.init_random(bits)
        b2 = sdmlib.Bitstring.init_from_bitstring(b1)
        b2.flip_random_bits(dist)
        assert b1.distance_to(b2) == dist

        h1 = set(address_space.scan_thread(b1, radius, 4))
        h2 = set(address_space.scan_thread(b2, radius, 4))

        y.append(len(h1&h2))
    return x, y
```

In [6]:
```
v = []
```

In [7]:
```
v.append((451, run(451)))
```

```
Distance: 1000 (100.00%)

In [10]: v.append((480, run(480)))

Distance: 1000 (100.00%)

In [9]: v.append((500, run(500)))

Distance: 1000 (100.00%)

In [14]: plt.figure(figsize=(8, 6), dpi=100)
         plt.hold(True)
         for d, points in v[:-1]:
             x, y = points
             ymax = max(y)
             y = [float(a)/ymax for a in y]
             plt.plot(x, y, label='Radius: {}'.format(d))
         plt.xlabel('Distance between two random bitstrings')
         plt.ylabel('Intersection of their circles')
         plt.legend()
         plt.hold(False)

/Library/Python/2.7/site-packages/ipykernel_launcher.py:2: MatplotlibDeprecationWarning: pyplot.hold
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.

/Library/Python/2.7/site-packages/ipykernel_launcher.py:11: MatplotlibDeprecationWarning: pyplot.hold
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.
  # This is added back by InteractiveShellApp.init_path()
```



1.3.4 Kanerva's Table 7.3 (page 70)

```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from IPython.display import clear_output
        %matplotlib inline
```

```
In [2]: bits = 1000
        sample = 1000000
        radius = 451
        scanner_type = sdmlib.SDM_SCANNER_OPENCL
```

```
In [3]: #address_space = sdmlib.AddressSpace.init_from_b64_file('sdm-10000w.as')
        #counter = sdmlib.Counter.load_file('sdm-10000w')
        address_space = sdmlib.AddressSpace.init_random(bits, sample)
        counter = sdmlib.Counter.init_zero(bits, sample)
        sdm = sdmlib.SDM(address_space, counter, radius, scanner_type)
```

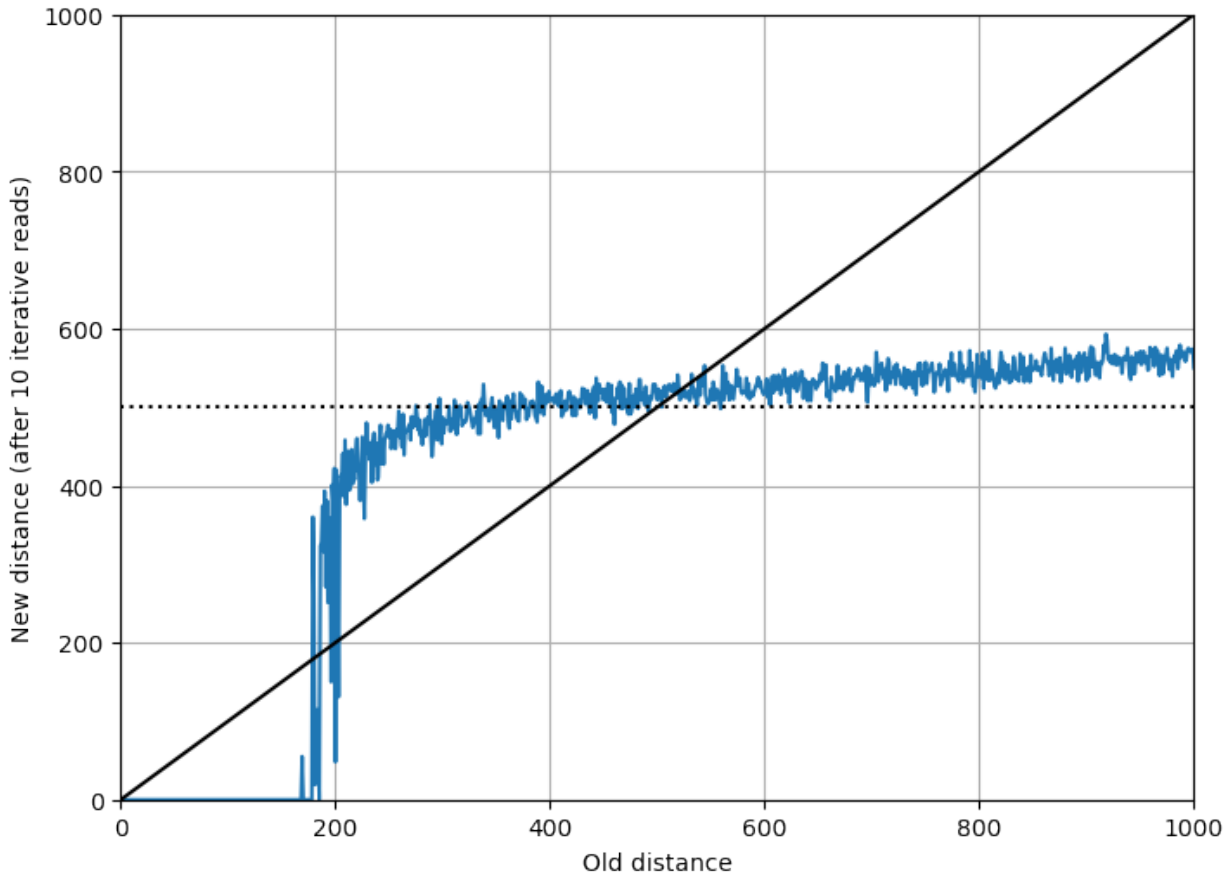
```
In [4]: for i in range(10000):
            clear_output(wait=True)
            print i
            bs = sdmlib.Bitstring.init_random(1000)
            sdm.write(bs, bs)
```

```
9999
```

```
In [5]: b = sdmlib.Bitstring.init_random(1000)
        sdm.write(b, b)

In [8]: from IPython.display import clear_output

        max_iter = 10
        samples = 1

        distances = []
        x = range(0, 1001)
        for i, dist in enumerate(x):
            clear_output(wait=True)
            print 'Distance: {:4d} ({:.2f}%)'.format(dist, 100.*(i+1)/len(x))
            v = []
            for j in range(samples):
                c = sdmlib.Bitstring.init_from_bitstring(b)
                c.flip_random_bits(dist)
                assert c.distance_to(b) == dist
                d = sdm.iter_read(c, max_iter=max_iter)
                v.append(d.distance_to(b))
            distances.append(1.0*sum(v)/len(v))
        print 'Done!'
```

```
Distance: 1000 (100.00%)
Done!
```

```
In [9]: plt.figure(figsize=(8, 6), dpi=100)
        plt.plot(x, distances)
        plt.plot(x, x, 'k')
        plt.plot(x, [500]*len(x), 'k:')
        #plt.title('Kanerva\'s Figure 7.3')
        if max_iter == 1:
            plt.ylabel('New distance (after a single read)')
        else:
            plt.ylabel('New distance (after {} iterative reads)'.format(max_iter))
        plt.xlabel('Old distance')
        plt.grid()
        #plt.axis([0, 1000, 0, 1000]);
        plt.axis([x[0], x[-1], x[0], x[-1]]);
```




```
In [ ]: c = sdmlib.Bitstring.init_from_bitstring(b)
        c.flip_random_bits(1000)
        d = c
        print 0, b.distance_to(d)
        for i in xrange(10):
            d = sdm.read(d)
            print i+1, b.distance_to(d)
```

1.3.5 Noise filter

```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from PIL import Image, ImageDraw, ImageFont
        import urllib, cStringIO
        import random
        from IPython.core.display import display, Image as IPythonImage
        %matplotlib inline
```

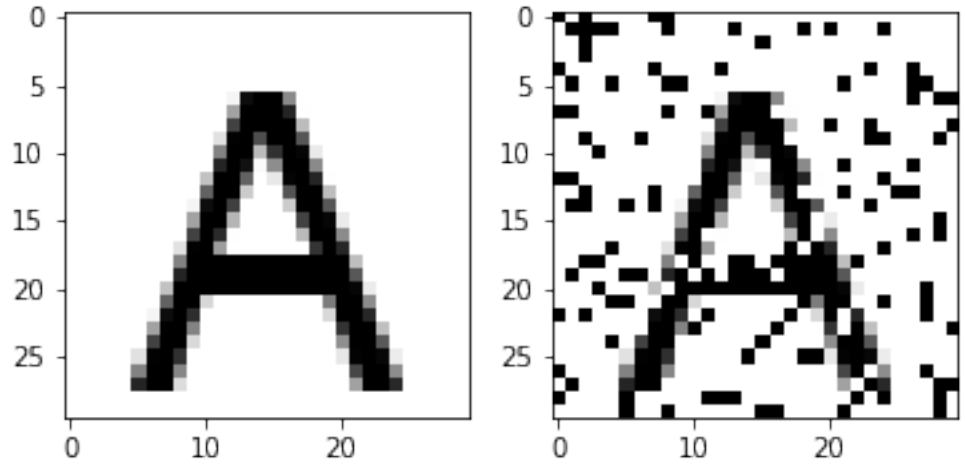
```
In [2]: width = 30
        height = 30
        noise_flip = True
```

```
In [3]: def gen_img(letter='A'):
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            font = ImageFont.truetype('Arial.ttf', 30)
            draw = ImageDraw.Draw(img)
```

```
            draw.text((5, 0), letter, (0, 0, 0), font=font)
            return img
```

```
In [4]: def gen_noise_add(img, p=0.15, flip=False):
            img2 = img.copy()
            draw = ImageDraw.Draw(img2)
            for py in xrange(height):
                for px in xrange(width):
                    if random.random() < p:
                        if flip:
                            pixel = img.getpixel((px, py))
                            value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                            assert value == 0 or value == 1
                            value = (1 - value)*255
                            draw.point((px, py), fill=(value, value, value))
                        else:
                            draw.point((px, py), fill=(0, 0, 0))
            return img2
```

```
In [5]: img = gen_img();
        img2 = gen_noise_add(img, flip=noise_flip)
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.subplot(1, 2, 2)
        plt.imshow(img2);
```




```
In [6]: def to_bitstring(img):
            v = []
            bs = sdmlib.Bitstring.init_ones(1000)
            for py in xrange(height):
                for px in xrange(width):
                    pixel = img.getpixel((px, py))
                    value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                    assert value == 0 or value == 1
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    bs.set_bit(idx, value)
                    v.append(value)
            v2 = [bs.get_bit(i) for i in xrange(height*width)]
            assert v == v2
            return bs
```

```
In [7]: def to_img(bs):
```

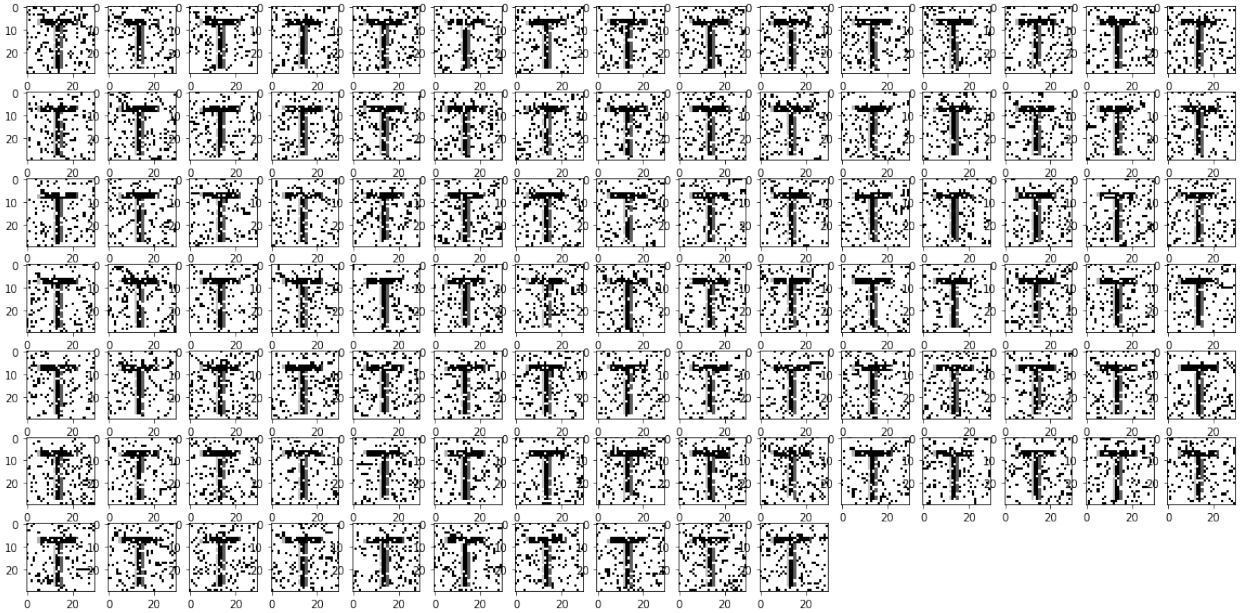
```
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            draw = ImageDraw.Draw(img)
            for py in xrange(height):
                for px in xrange(width):
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    x = 255*bs.get_bit(idx)
                    draw.point((px, py), fill=(x, x, x))
            return img
```

```
In [8]: bits = 1000
        sample = 1000000
        scanner_type = sdmlib.SDM_SCANNER_OPENCL
```

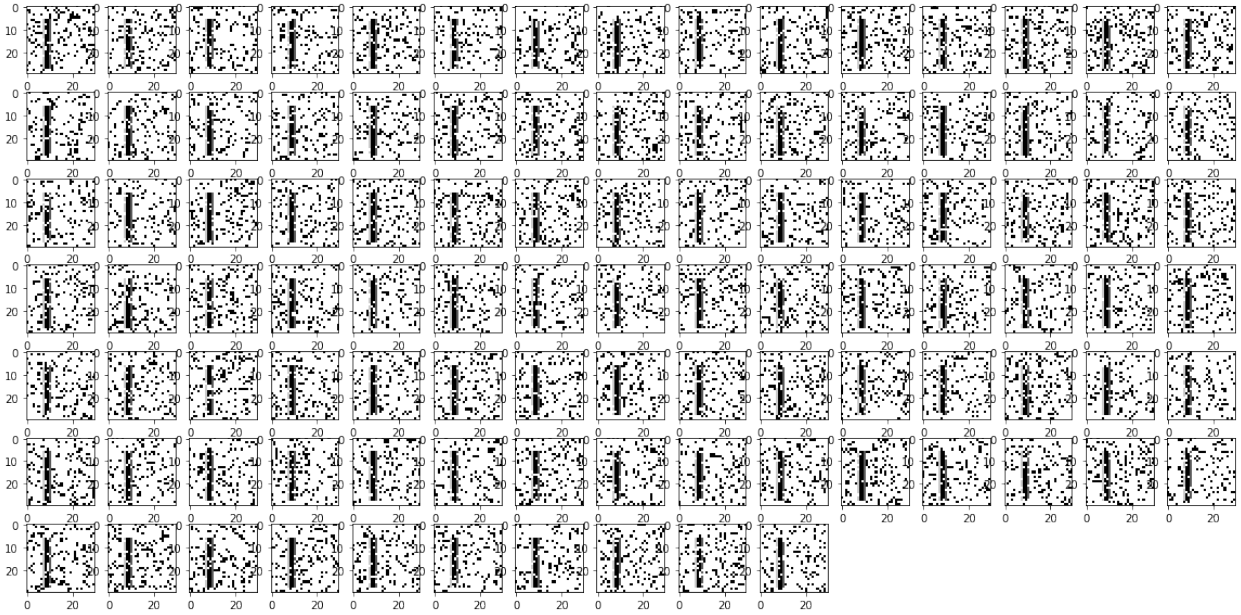
```
In [9]: address_space = sdmlib.AddressSpace.init_random(bits, sample)
        counter = sdmlib.Counter.init_zero(bits, sample)
        sdm = sdmlib.SDM(address_space, counter, 451, scanner_type)
```

```
In [10]: def fill_memory(letter, p=0.15, n=100):
             cols = 15
             rows = n//cols + 1
             plt.figure(figsize=(20,10))
             for i in xrange(n):
                 img = gen_img(letter=letter);
                 img2 = gen_noise_add(img, flip=noise_flip)
                 #display(img2)
                 plt.subplot(rows, cols, i+1)
                 plt.imshow(img2)
                 bs = to_bitstring(img2)
                 sdm.write(bs, bs)
```

```
In [11]: fill_memory('T')
```

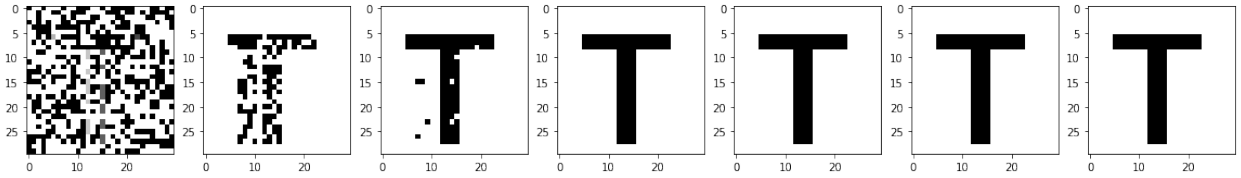


```
In [12]: fill_memory('I')
```

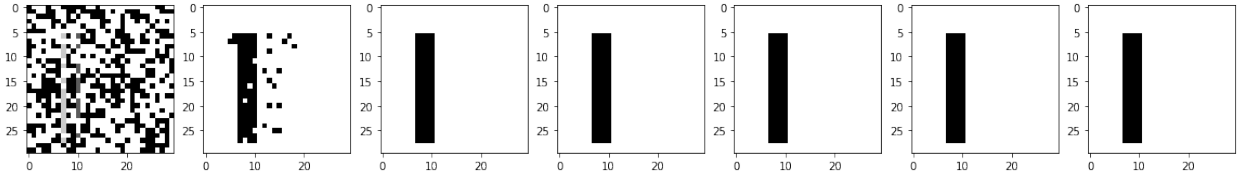


```
In [13]: def read(letter, n=6, p=0.25):
             n = 6
             cols = 7
             rows = n//cols + 1
             plt.figure(figsize=(20,10))

             img = gen_img(letter=letter);
             img2 = gen_noise_add(img, p=p, flip=noise_flip)
             plt.subplot(rows, cols, 1)
             plt.imshow(img2)

             for i in xrange(n):
                 bs2 = to_bitstring(img2)
                 bs3 = sdm.read(bs2)
                 img3 = to_img(bs3)
                 plt.subplot(rows, cols, i+2)
                 plt.imshow(img3)
                 img2 = img3
```

```
In [20]: read('T', p=0.42)
```



```
In [15]: read('I', p=0.42)
```



1.3.6 Classification Test #1

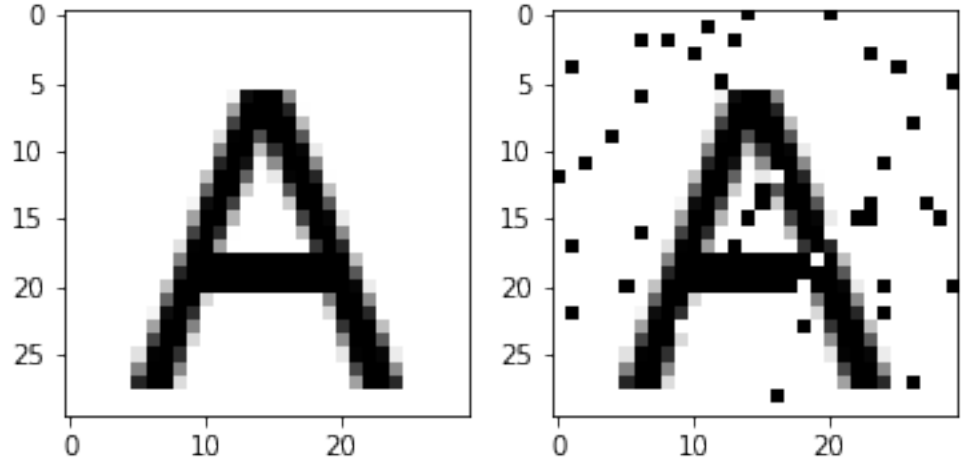
```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from PIL import Image, ImageDraw, ImageFont
        import urllib, cStringIO
        import random
        from IPython.core.display import display, Image as IPythonImage
        %matplotlib inline
```

```
In [2]: width = 30
        height = 30
        noise_flip = True
```

```
In [3]: def gen_img(letter='A'):
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            font = ImageFont.truetype('Arial.ttf', 30)
            draw = ImageDraw.Draw(img)
            draw.text((5, 0), letter, (0, 0, 0), font=font)
            return img
```

```
In [4]: def gen_noise_add(img, p=0.15, flip=False):
            img2 = img.copy()
            draw = ImageDraw.Draw(img2)
            for py in xrange(height):
                for px in xrange(width):
                    if random.random() < p:
                        if flip:
                            pixel = img.getpixel((px, py))
                            value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                            assert value == 0 or value == 1
                            value = (1 - value)*255
                            draw.point((px, py), fill=(value, value, value))
                        else:
                            draw.point((px, py), fill=(0, 0, 0))
            return img2
```

```
In [5]: img = gen_img();
        img2 = gen_noise_add(img, p=0.05, flip=noise_flip)
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.subplot(1, 2, 2)
        plt.imshow(img2);
```



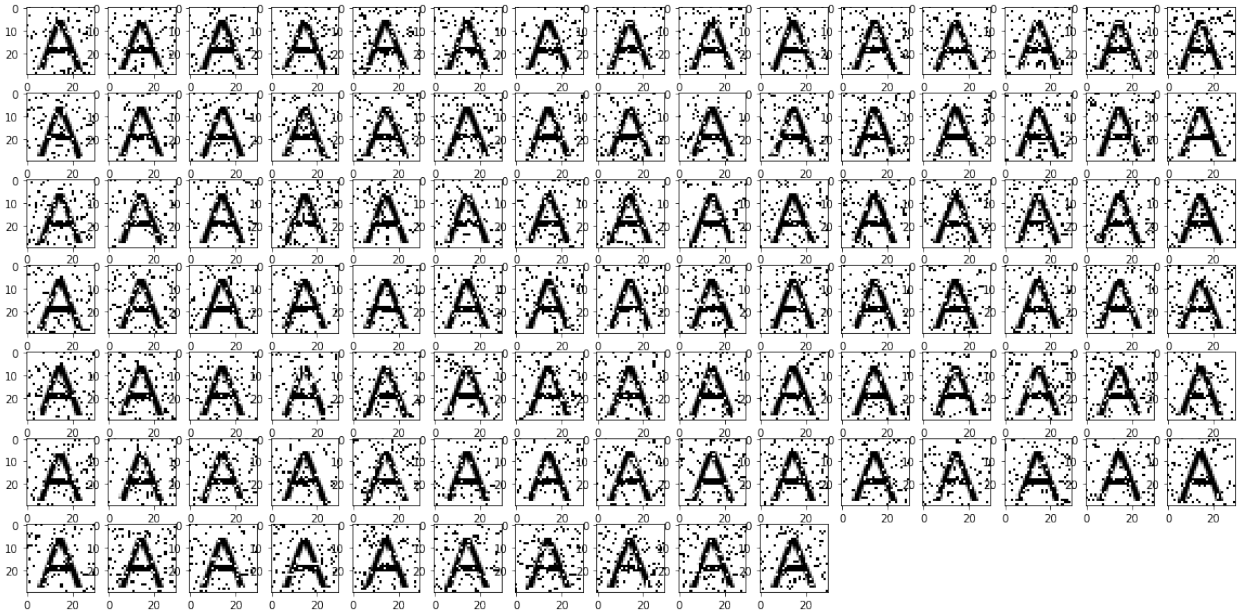
```
In [6]: def to_bitstring(img):
            v = []
            bs = sdmlib.Bitstring.init_ones(1000)
            for py in xrange(height):
                for px in xrange(width):
                    pixel = img.getpixel((px, py))
                    value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                    assert value == 0 or value == 1
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    bs.set_bit(idx, value)
                    v.append(value)
            v2 = [bs.get_bit(i) for i in xrange(height*width)]
            assert v == v2
            return bs

In [7]: def to_img(bs):
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            draw = ImageDraw.Draw(img)
            for py in xrange(height):
                for px in xrange(width):
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    x = 255*bs.get_bit(idx)
                    draw.point((px, py), fill=(x, x, x))
            return img

In [8]: bits = 1000
        sample = 1000000
        scanner_type = sdmlib.SDM_SCANNER_THREAD

In [9]: address_space = sdmlib.AddressSpace.init_from_b64_file('sdm-letters.as')
        counter = sdmlib.Counter.create_file('sdm-classification', bits, sample)
        sdm = sdmlib.SDM(address_space, counter, 451, scanner_type)

In [10]: for i in xrange(100):
             print i,
             b = sdmlib.Bitstring.init_random(1000)
             sdm.write(b, b)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

In [11]: def fill_memory(letter, label_bs, p=0.1, n=100):
             cols = 15
             rows = n//cols + 1
             plt.figure(figsize=(20,10))
             for i in xrange(n):
                 img = gen_img(letter=letter);
                 img2 = gen_noise_add(img, p=p, flip=noise_flip)
                 #display(img2)
                 plt.subplot(rows, cols, i+1)
                 plt.imshow(img2)
                 bs = to_bitstring(img2)
                 bs.xor(label_bs)
                 sdm.write(bs, bs)
             plt.show()

In [12]: def read(letter, label_bs, n=6, p=0.2, radius=None):
             n = 7
             cols = 15
             rows = n//cols + 1
             plt.figure(figsize=(20,10))
```

```
            img = gen_img(letter=letter);
            img2 = gen_noise_add(img, p=p, flip=noise_flip)
            plt.subplot(rows, cols, 1)
            plt.imshow(img2)

            for i in xrange(n):
                bs2 = to_bitstring(img2)
                bs2.xor(label_bs)
                bs3 = sdm.read(bs2, radius=radius)
                if bs3 == bs2:
                    break
                bs3.xor(label_bs)
                img3 = to_img(bs3)
                plt.subplot(rows, cols, i+2)
                plt.imshow(img3)
                img2 = img3
```

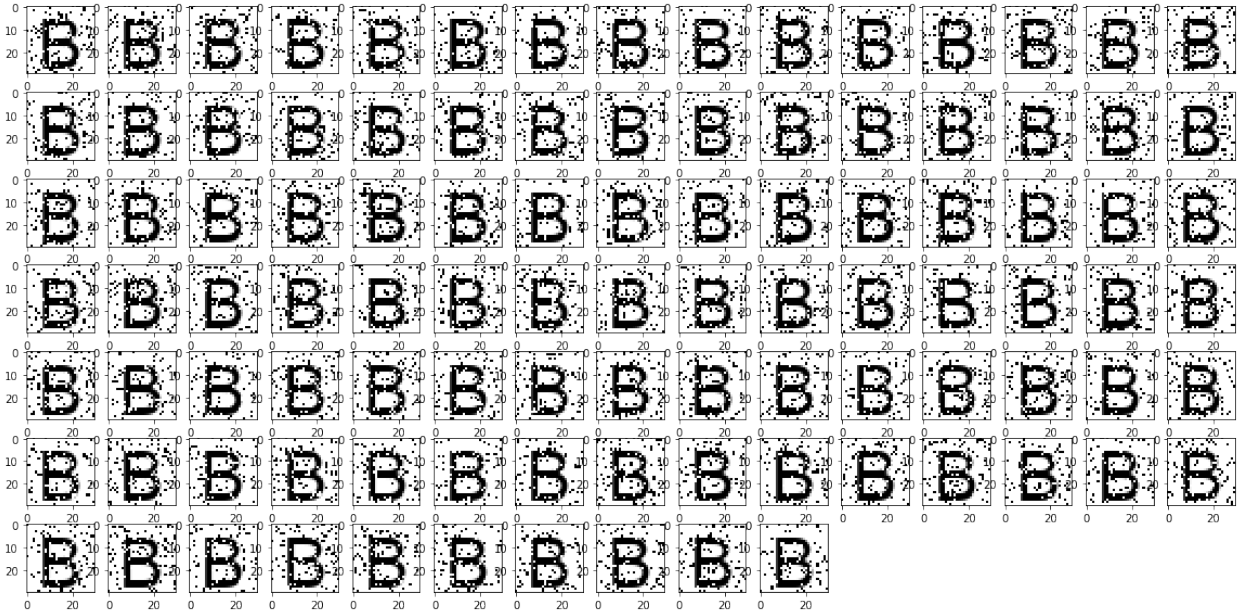
In [13]:
```
labels = list('ABCD8OQ')
label_to_bs = {}
for x in labels:
    label_to_bs[x] = sdmlib.Bitstring.init_random(1000)
```

In [14]:
```
for x in labels:
    print 'Training for label {}...'.format(x)
    fill_memory(x, label_to_bs[x])
```

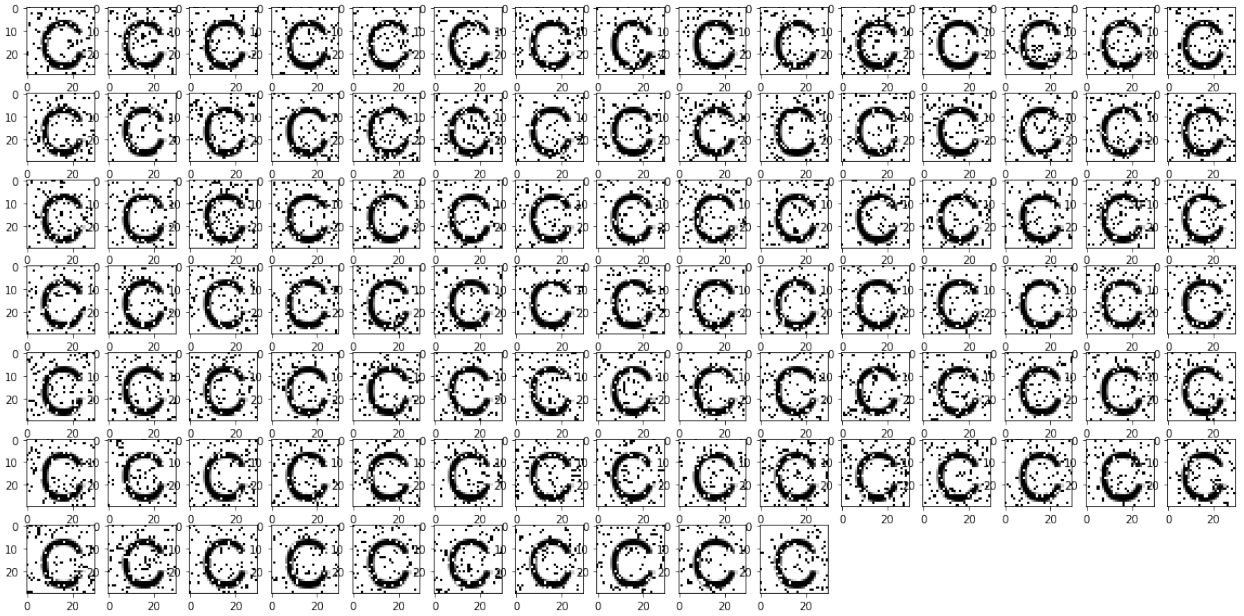
```
Training for label A...
```



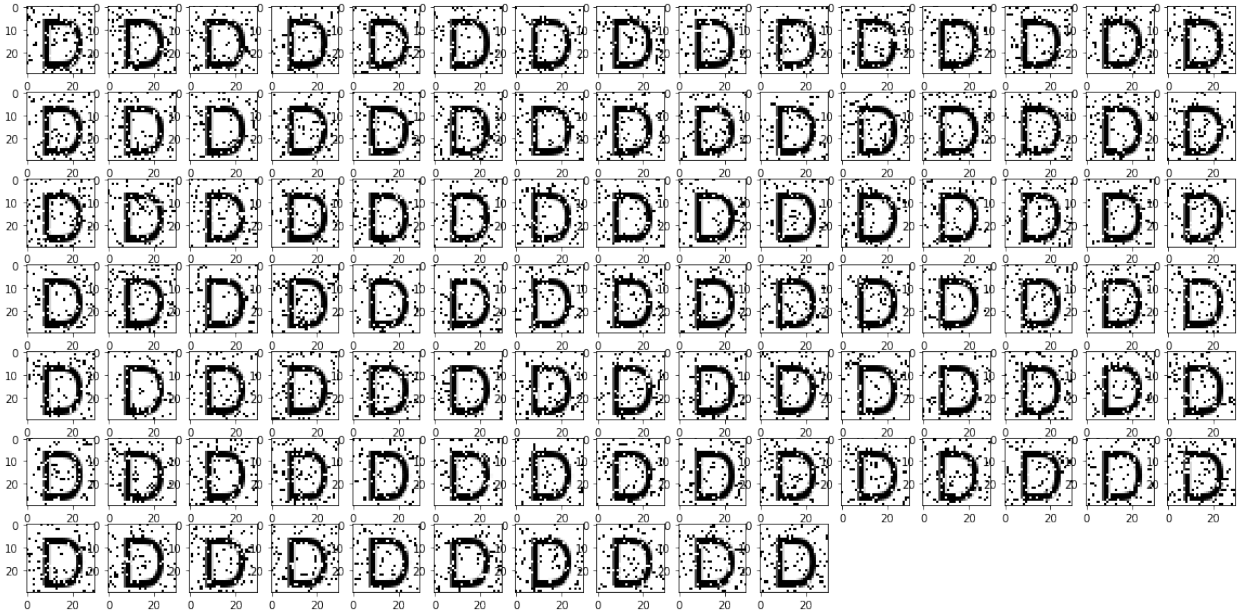
```
Training for label B...
```



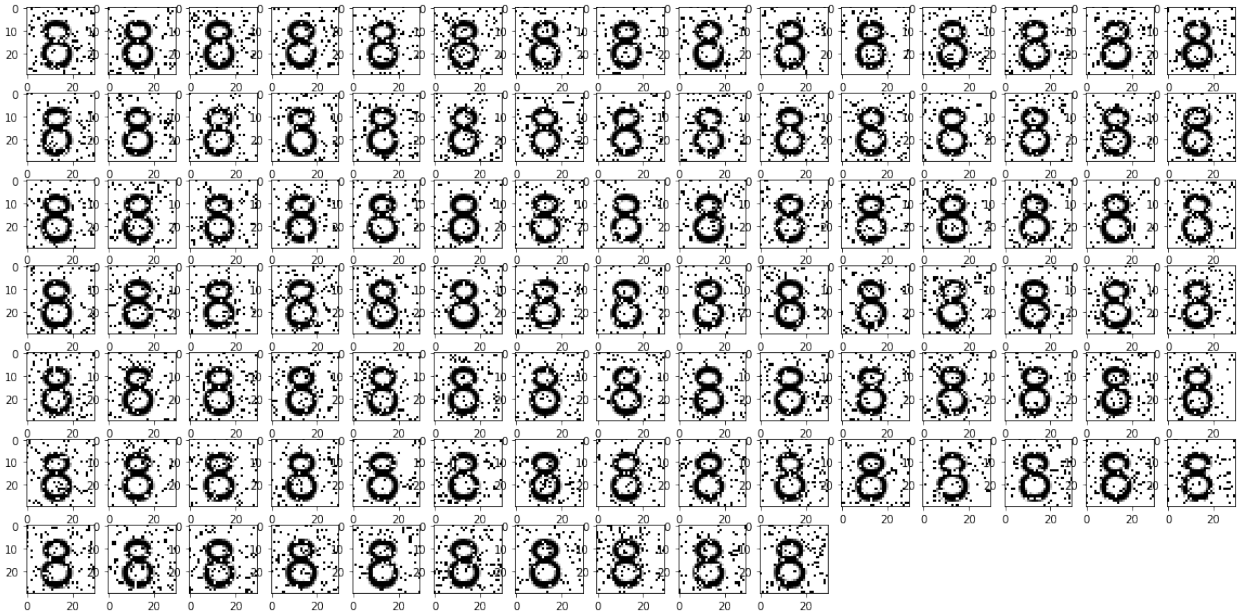
```
Training for label C...
```



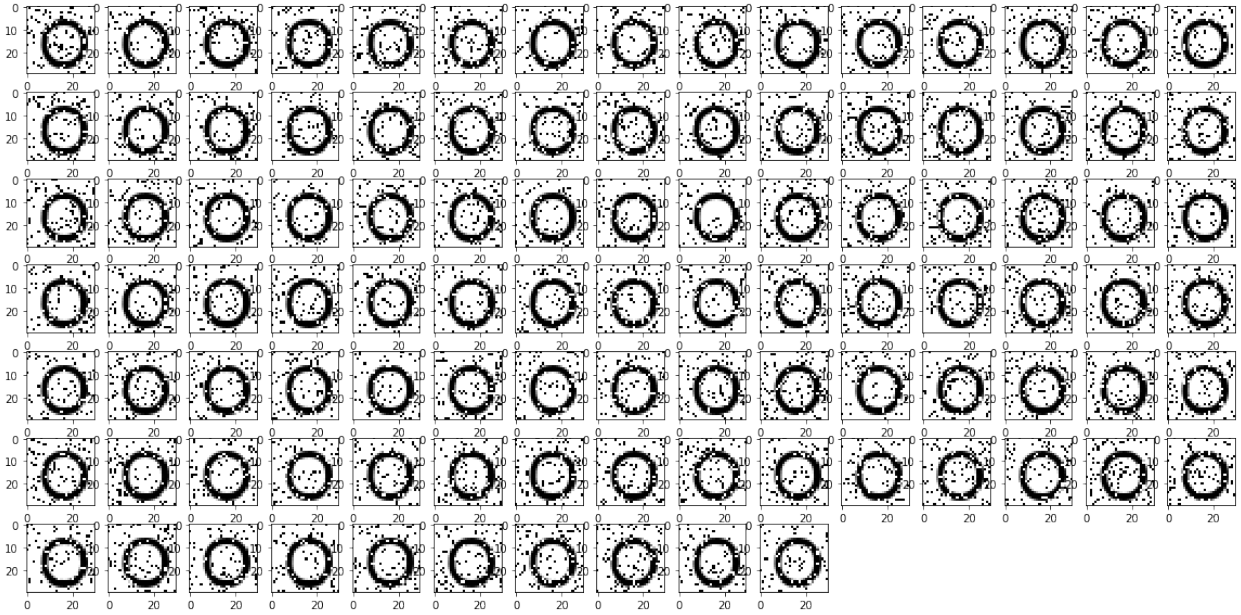
```
Training for label D...
```



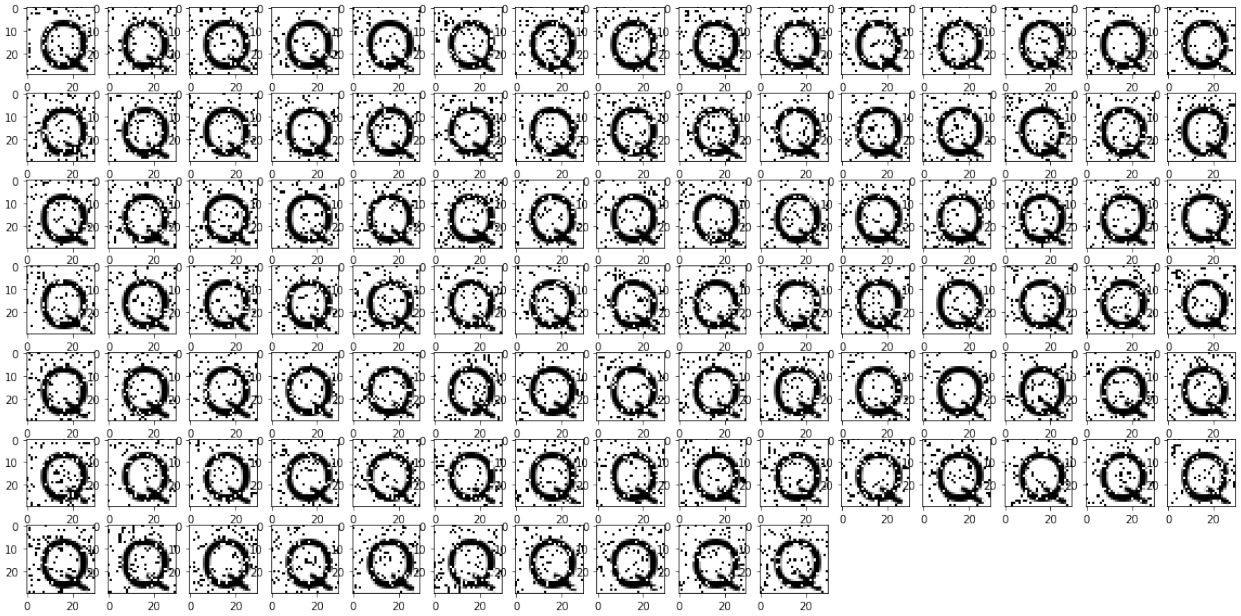
```
Training for label 8...
```



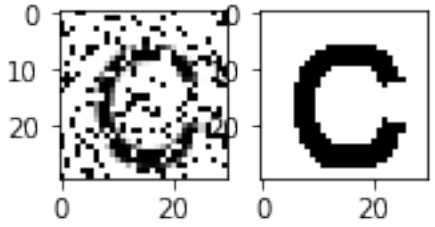
```
Training for label O...
```

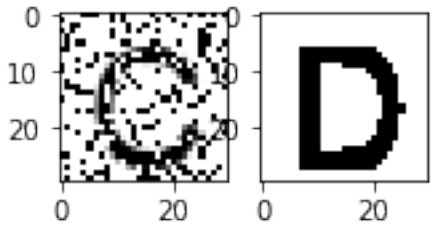


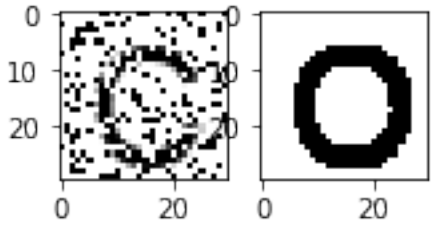
```
Training for label Q...
```



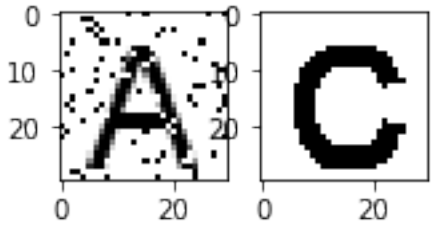
```
In [15]: read('C', label_to_bs['C'])
         read('C', label_to_bs['D'])
         read('C', label_to_bs['O'])
```







```
In [16]: read('A', label_to_bs['C'], p=0.1)
```



```
In [17]: read('A', sdmlib.Bitstring.init_random(1000), p=0)
```

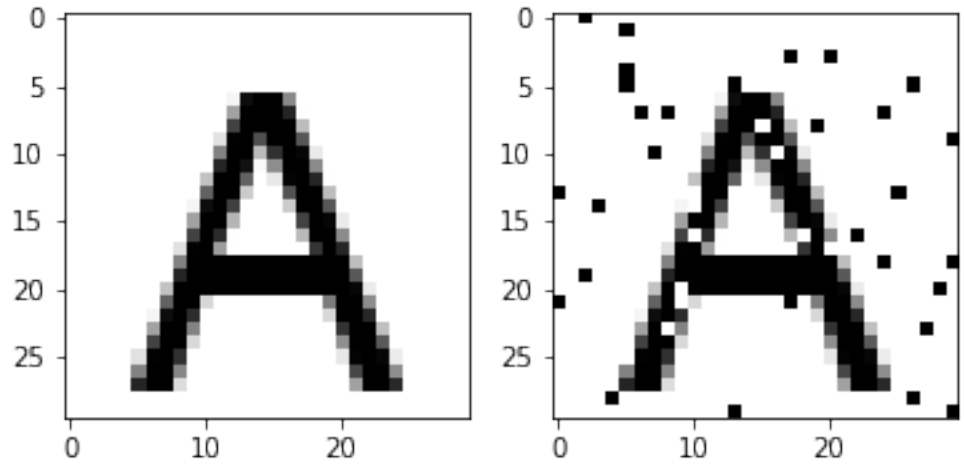
```
In [18]: def intersection(a, b):
             bs1 = to_bitstring(gen_img(letter=a))
             bs2 = to_bitstring(gen_img(letter=b))
             hl1 = set(address_space.scan_thread(bs1, 451, 4))
             hl2 = set(address_space.scan_thread(bs2, 451, 4))
             return len(hl1 & hl2)
         print intersection('B', '8')
```

```
200
```

1.3.7 Classification Test #2

```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from PIL import Image, ImageDraw, ImageFont
        import urllib, cStringIO
        import random
        from IPython.core.display import display, Image as IPythonImage
        %matplotlib inline
```

```
In [2]: width = 30
        height = 30
        noise_flip = True

In [3]: def gen_img(letter='A'):
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            font = ImageFont.truetype('Arial.ttf', 30)
            draw = ImageDraw.Draw(img)
            draw.text((5, 0), letter, (0, 0, 0), font=font)
            return img

In [4]: def gen_noise_add(img, p=0.15, flip=False):
            img2 = img.copy()
            draw = ImageDraw.Draw(img2)
            for py in xrange(height):
                for px in xrange(width):
                    if random.random() < p:
                        if flip:
                            pixel = img.getpixel((px, py))
                            value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                            assert value == 0 or value == 1
                            value = (1 - value)*255
                            draw.point((px, py), fill=(value, value, value))
                        else:
                            draw.point((px, py), fill=(0, 0, 0))
            return img2

In [5]: img = gen_img();
        img2 = gen_noise_add(img, p=0.05, flip=noise_flip)
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.subplot(1, 2, 2)
        plt.imshow(img2);
```




```
In [6]: def to_bitstring(img):
            v = []
            bs = sdmlib.Bitstring.init_ones(1000)
            for py in xrange(height):
                for px in xrange(width):
                    pixel = img.getpixel((px, py))
                    value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                    assert value == 0 or value == 1
                    idx = px+width*py
```

```
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    bs.set_bit(idx, value)
                    v.append(value)
            v2 = [bs.get_bit(i) for i in xrange(height*width)]
            assert v == v2
            return bs
```

In [7]:
```
def to_img(bs):
    img = Image.new('RGBA', (30, 30), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    for py in xrange(height):
        for px in xrange(width):
            idx = px+width*py
            assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
            x = 255*bs.get_bit(idx)
            draw.point((px, py), fill=(x, x, x))
    return img
```

In [8]:
```
bits = 1000
sample = 1000000
scanner_type = sdmlib.SDM_SCANNER_OPENCL
```

In [9]:
```
address_space = sdmlib.AddressSpace.init_from_b64_file('sdm-letters.as')
counter = sdmlib.Counter.create_file('sdm-classification-2', bits, sample)
sdm = sdmlib.SDM(address_space, counter, 451, scanner_type)
```

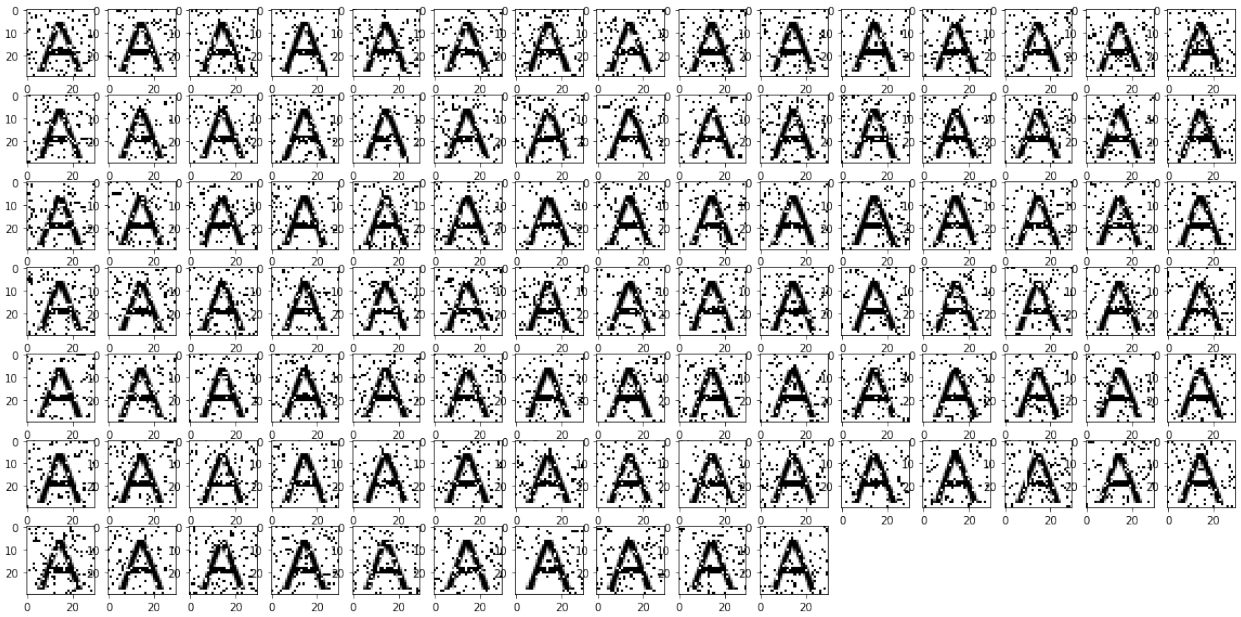
In [10]:
```
def fill_memory(letter, label_bs, p=0.1, n=100):
    cols = 15
    rows = n//cols + 1
    plt.figure(figsize=(20,10))
    for i in xrange(n):
        img = gen_img(letter=letter);
        img2 = gen_noise_add(img, p=p, flip=noise_flip)
        #display(img2)
        plt.subplot(rows, cols, i+1)
        plt.imshow(img2)
        bs = to_bitstring(img2)
        sdm.write(bs, label_bs)
    plt.show()
```

In [11]:
```
def read(letter, n=6, p=0.2, radius=None):
    n = 7
    cols = 15
    rows = n//cols + 1
    plt.figure(figsize=(20,10))

    img = gen_img(letter=letter);
    img2 = gen_noise_add(img, p=p, flip=noise_flip)
    plt.subplot(rows, cols, 1)
    plt.imshow(img2)

    bs2 = to_bitstring(img2)
    bs3 = sdm.read(bs2, radius=radius)

    label = min(label_to_bs.items(), key=lambda v: bs3.distance_to(v[1]))
    return label[0]
```

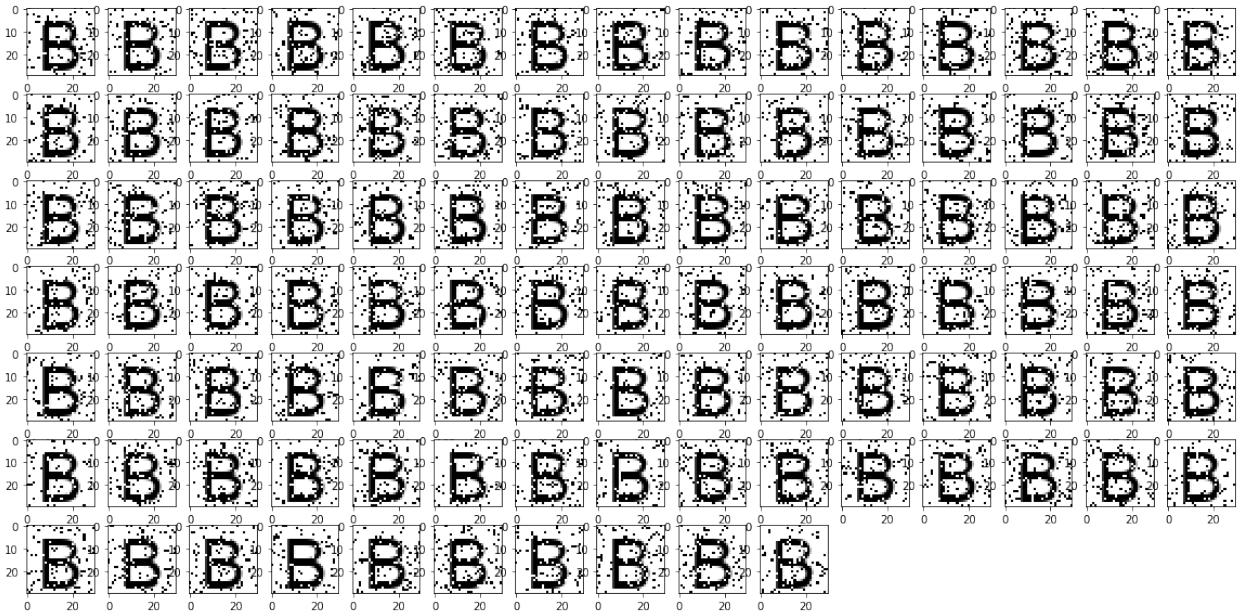
In [12]:
```
labels = list('ABCD8OQ0')
label_to_bs = {}
for x in labels:
    label_to_bs[x] = sdmlib.Bitstring.init_random(1000)
```

```
In [13]: for x in labels:
             print 'Training for label {}...'.format(x)
             fill_memory(x, label_to_bs[x])
```

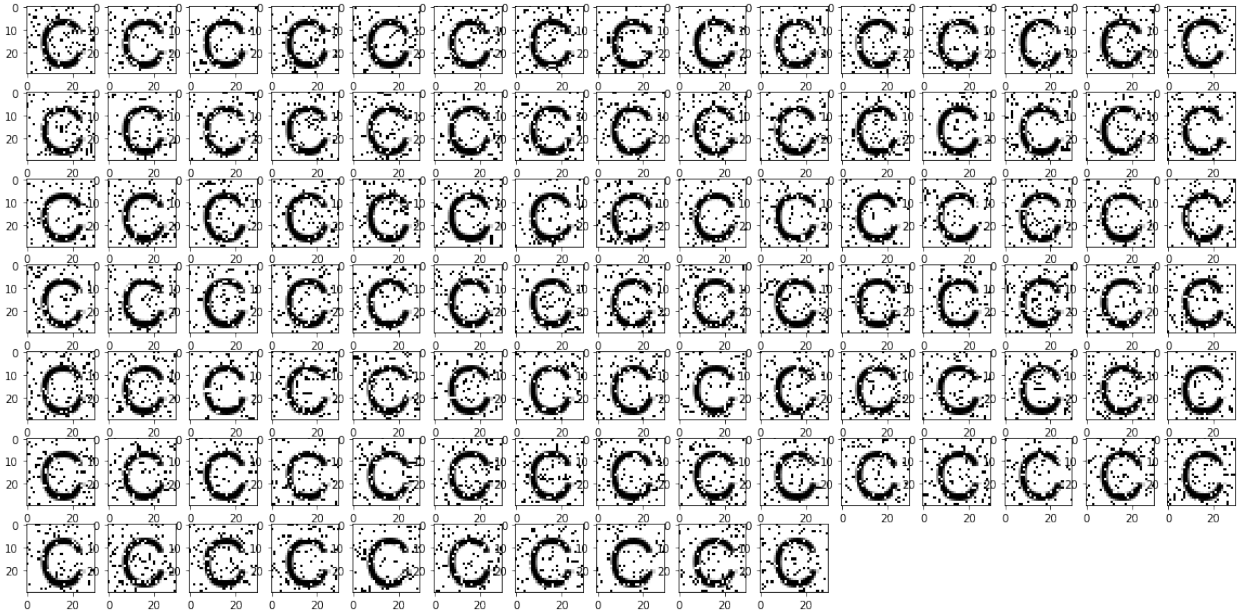
```
Training for label A...
```



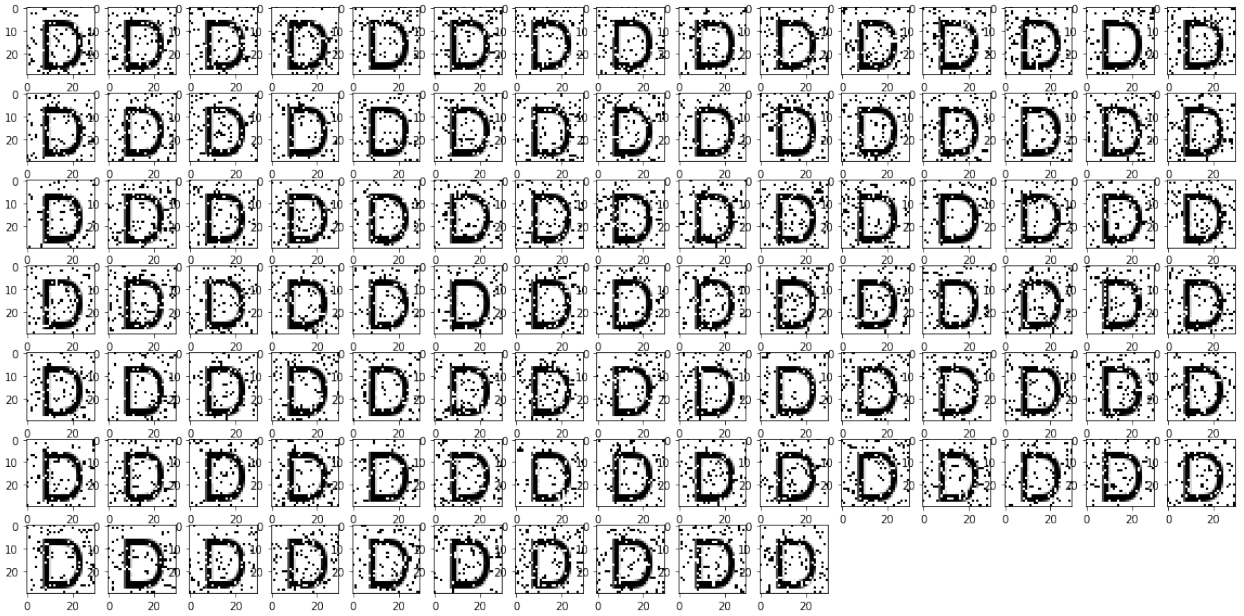
```
Training for label B...
```



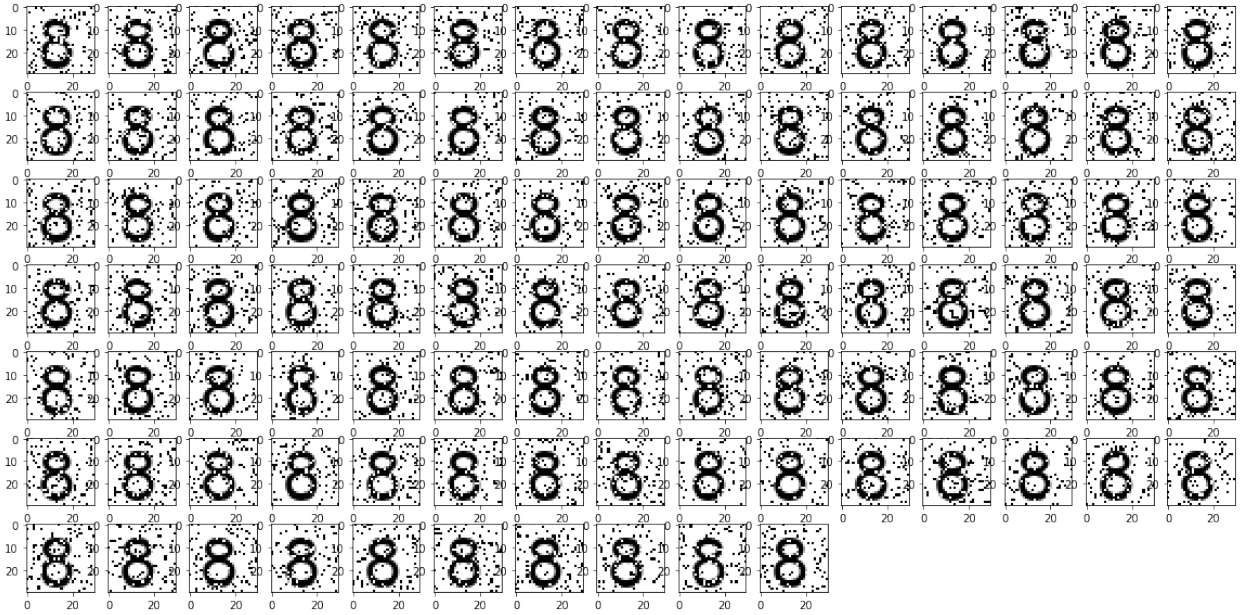
```
Training for label C...
```



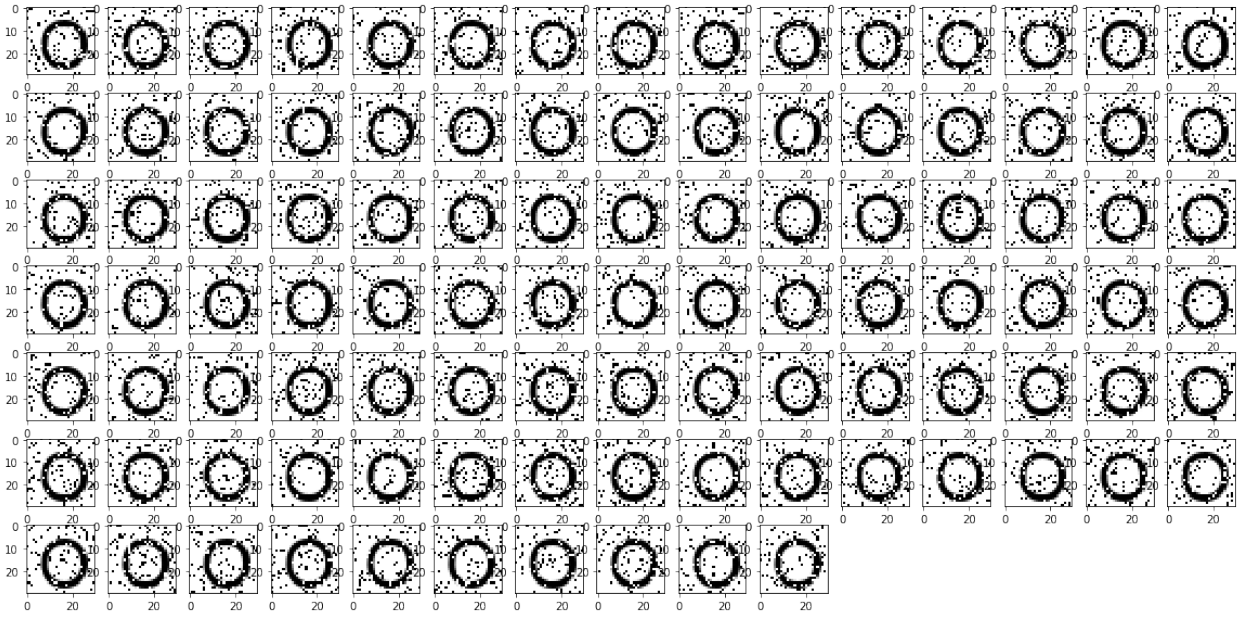
```
Training for label D...
```



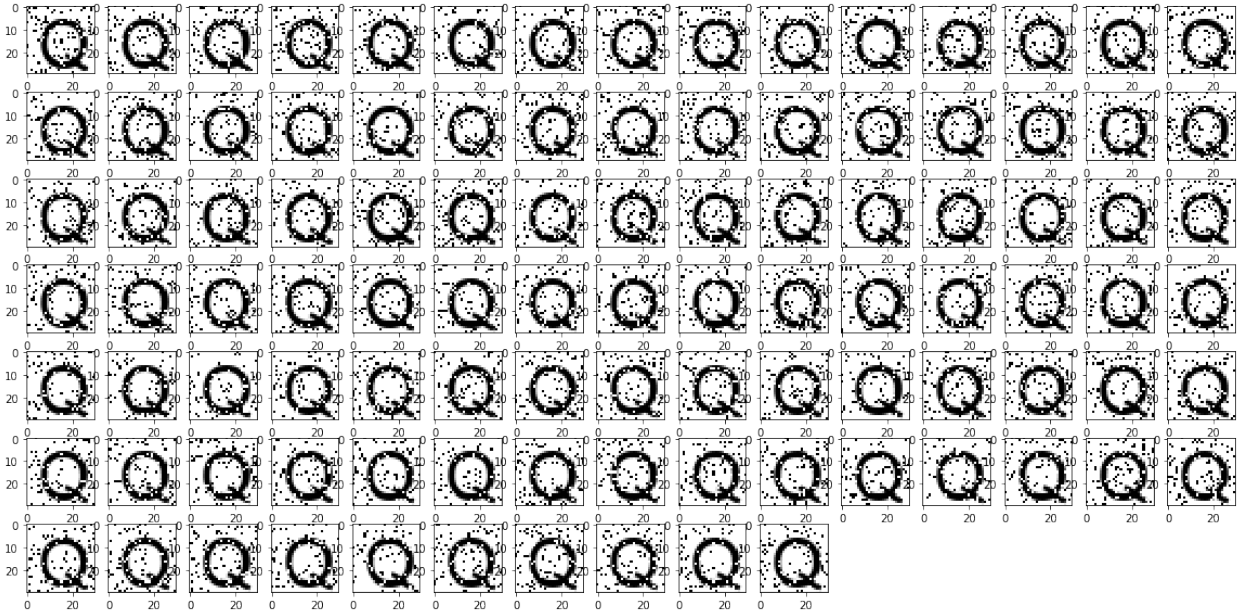
```
Training for label 8...
```



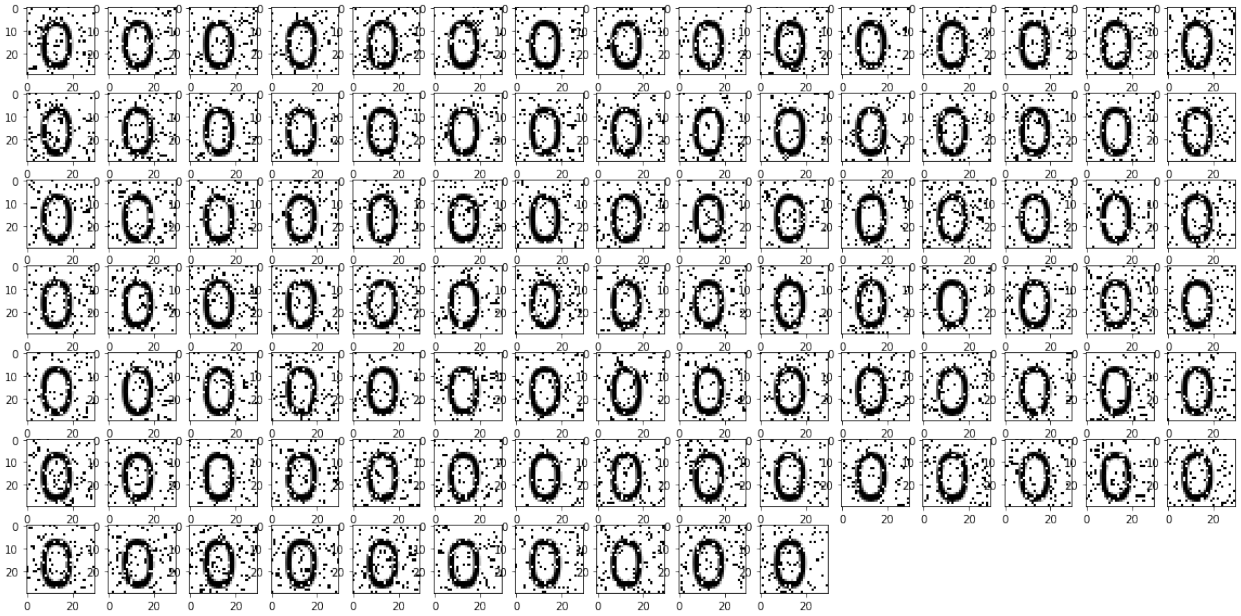
```
Training for label O...
```



```
Training for label Q...
```

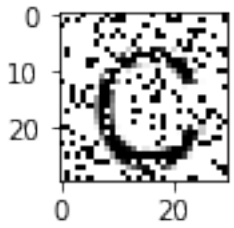


```
Training for label 0...
```



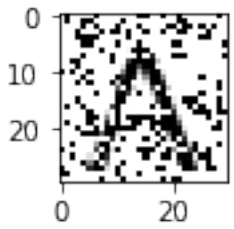
```
In [14]: read('C')
```

```
Out[14]: 'C'
```

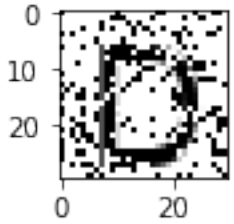


```
In [15]: read('A')
```

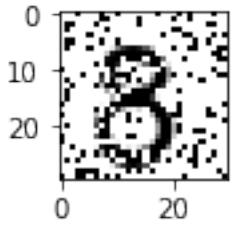
```
Out[15]: 'A'
```



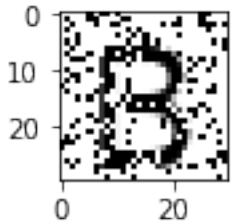

```
In [16]: read('D')
Out[16]: 'D'
```



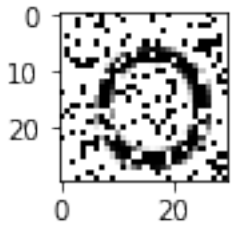

```
In [17]: read('8')
Out[17]: '8'
```




```
In [18]: read('B')
Out[18]: 'B'
```

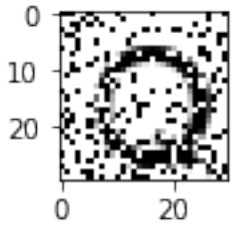



```
In [19]: read('O')
Out[19]: 'O'
```



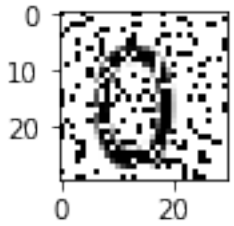

```
In [35]: read('Q')
```

```
Out[35]: 'Q'
```



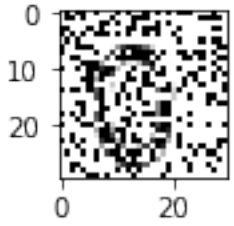

```
In [25]: read('0')
```

```
Out[25]: '0'
```




```
In [45]: read('0', p=0.3)
```

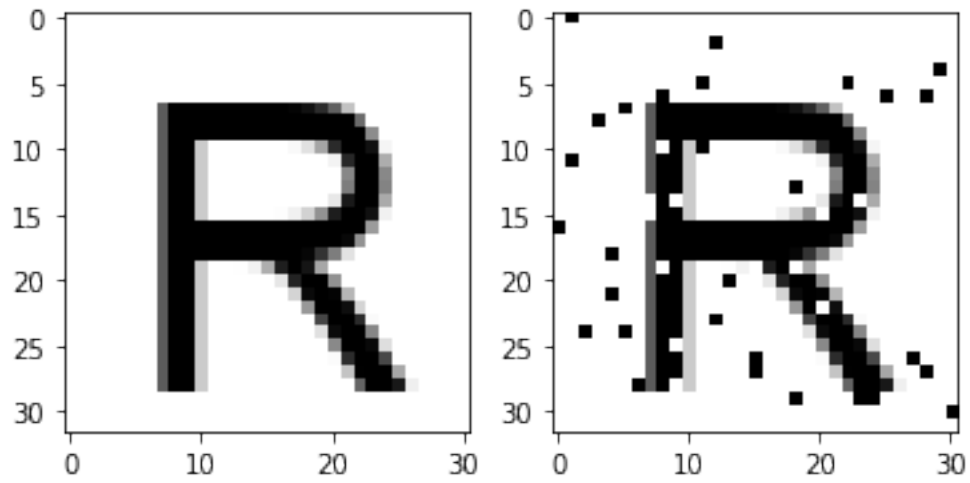
```
Out[45]: '0'
```



1.3.8 Classification Test #3

- *Generating labels*
- *Training the SDM*
- *Testing with high noise*
- *Testing with low noise*
- *Testing with no noise*

```
In [1]: import sdm as sdmlib
        import matplotlib.pyplot as plt
        from PIL import Image, ImageDraw, ImageFont
        import urllib, cStringIO
        import random
        from IPython.core.display import display, Image as IPythonImage
        %matplotlib inline

In [2]: width = 31
        height = 32
        noise_flip = True

In [3]: def gen_img(letter='A'):
            img = Image.new('RGBA', (width, height), (255, 255, 255))
            font = ImageFont.truetype('Arial.ttf', 30)
            draw = ImageDraw.Draw(img)
            w, h = draw.textsize(letter, font=font)
            top = (height-w)//2
            left = (width-h)//2 if h <= 30 else 30-2-h
            draw.text((top, left), letter, (0, 0, 0), font=font)
            return img

In [4]: def gen_noise_add(img, p=0.15, flip=False):
            img2 = img.copy()
            draw = ImageDraw.Draw(img2)
            for py in xrange(height):
                for px in xrange(width):
                    if random.random() < p:
                        if flip:
                            pixel = img.getpixel((px, py))
                            value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                            assert value == 0 or value == 1
                            value = (1 - value)*255
                            draw.point((px, py), fill=(value, value, value))
                        else:
                            draw.point((px, py), fill=(0, 0, 0))
            return img2

In [5]: img = gen_img(letter='R');
        img2 = gen_noise_add(img, p=0.05, flip=noise_flip)
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.subplot(1, 2, 2)
        plt.imshow(img2);
```




```
In [6]: def to_bitstring(img):
            v = []
            bs = sdmlib.Bitstring.init_ones(1000)
            for py in xrange(height):
                for px in xrange(width):
                    pixel = img.getpixel((px, py))
                    value = sum([int(x/255+0.5) for x in pixel[:3]])//3
                    assert value == 0 or value == 1
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    bs.set_bit(idx, value)
                    v.append(value)
            v2 = [bs.get_bit(i) for i in xrange(height*width)]
            assert v == v2
            return bs
```

```
In [7]: def to_img(bs):
            img = Image.new('RGBA', (30, 30), (255, 255, 255))
            draw = ImageDraw.Draw(img)
            for py in xrange(height):
                for px in xrange(width):
                    idx = px+width*py
                    assert idx >= 0 and idx < 1000, 'Ops {} {} {}'.format(x, y, idx)
                    x = 255*bs.get_bit(idx)
                    draw.point((px, py), fill=(x, x, x))
            return img
```

```
In [8]: bits = 1000
        sample = 1000000
        scanner_type = sdmlib.SDM_SCANNER_OPENCL
```

```
In [9]: address_space = sdmlib.AddressSpace.init_from_b64_file('sdm-letters.as')
```

```
In [10]: counter = sdmlib.Counter.create_file('sdm-classification-3', bits, sample)
         sdm = sdmlib.SDM(address_space, counter, 451, scanner_type)
```

```
In [11]: def fill_memory(letter, label_bs, p=0.1, n=100):
             cols = 15
             rows = n//cols + 1
             plt.figure(figsize=(20,10))
             for i in xrange(n):
                 img = gen_img(letter=letter);
                 img2 = gen_noise_add(img, p=p, flip=noise_flip)
```

```
                #display(img2)
                plt.subplot(rows, cols, i+1)
                plt.imshow(img2)
                bs = to_bitstring(img2)
                sdm.write(bs, label_bs)
            plt.show()
```

```
In [12]: def read(letter, n=6, p=0.2, radius=None):
            img = gen_img(letter=letter);
            img2 = gen_noise_add(img, p=p, flip=noise_flip)
            plt.imshow(img2)

            bs2 = to_bitstring(img2)
            bs3 = sdm.read(bs2, radius=radius)

            label = min(label_to_bs.items(), key=lambda v: bs3.distance_to(v[1]))
            return label[0]
```

```
In [13]: def iter_read(letter, n=10, p=0.2):
            cols = 15
            rows = n//cols + 1
            plt.figure(figsize=(20,10))

            wrong = 0
            correct = 0
            answers = []
            for i in xrange(n):
                plt.subplot(rows, cols, i+1)
                y = read(x, p=p)
                answers.append(y)
                if x == y:
                    correct += 1
                else:
                    wrong += 1
            plt.show()
            print '!! {} correct={:2d} wrong={:2d} answers={}'.format(x, correct, wrong, answers)
```

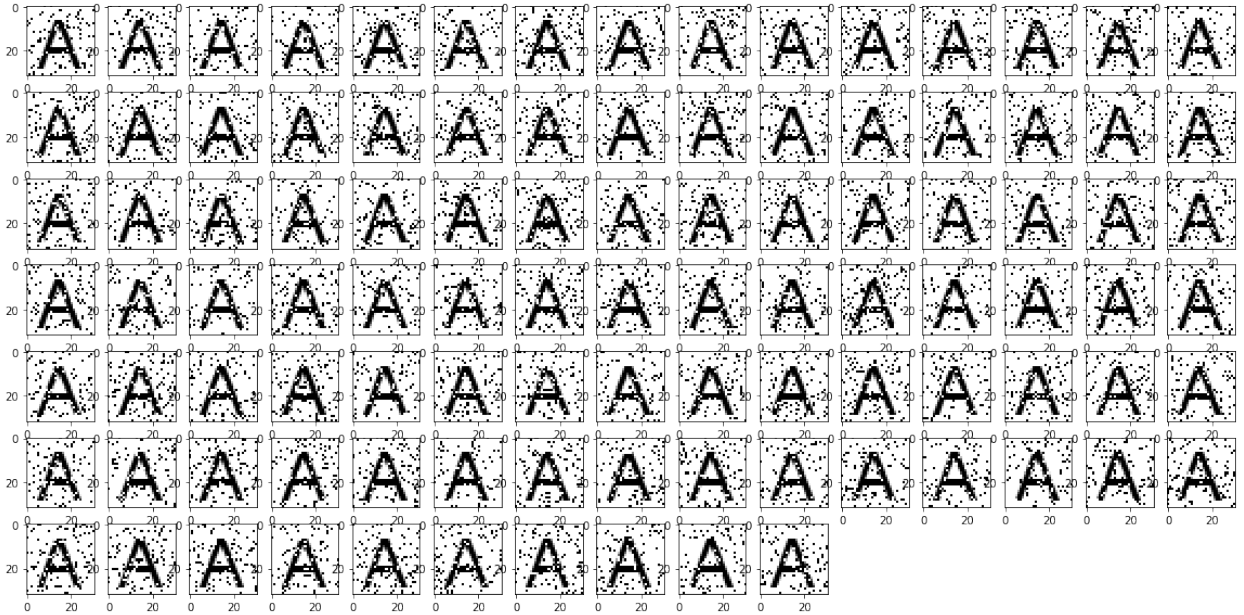
Generating labels

```
In [14]: labels = list('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789')
         label_to_bs = {}
         for x in labels:
             label_to_bs[x] = sdmlib.Bitstring.init_random(1000)
```

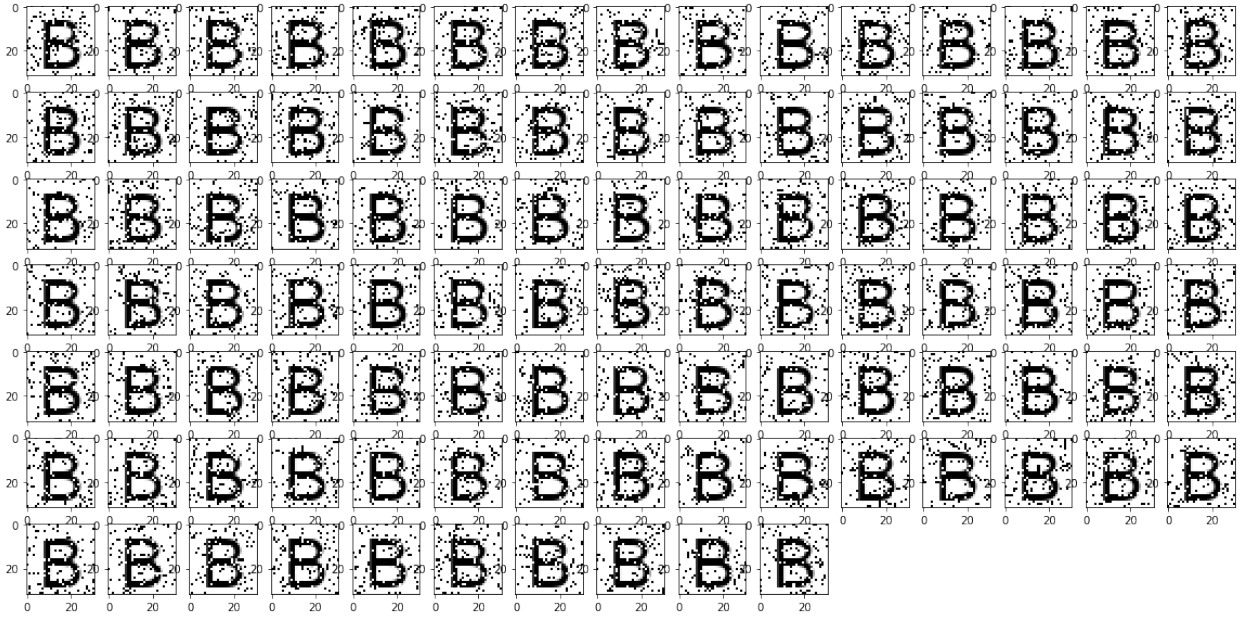
Training the SDM

```
In [15]: for x in labels:
             print 'Training for label {}...'.format(x)
             fill_memory(x, label_to_bs[x])
```

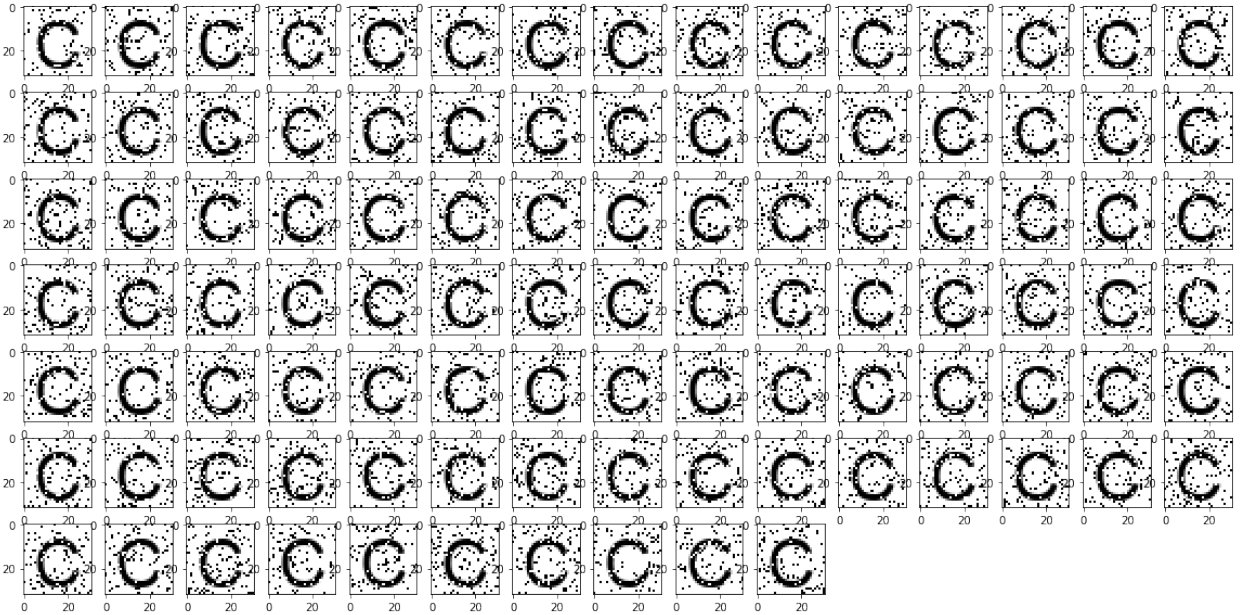
```
Training for label A...
```



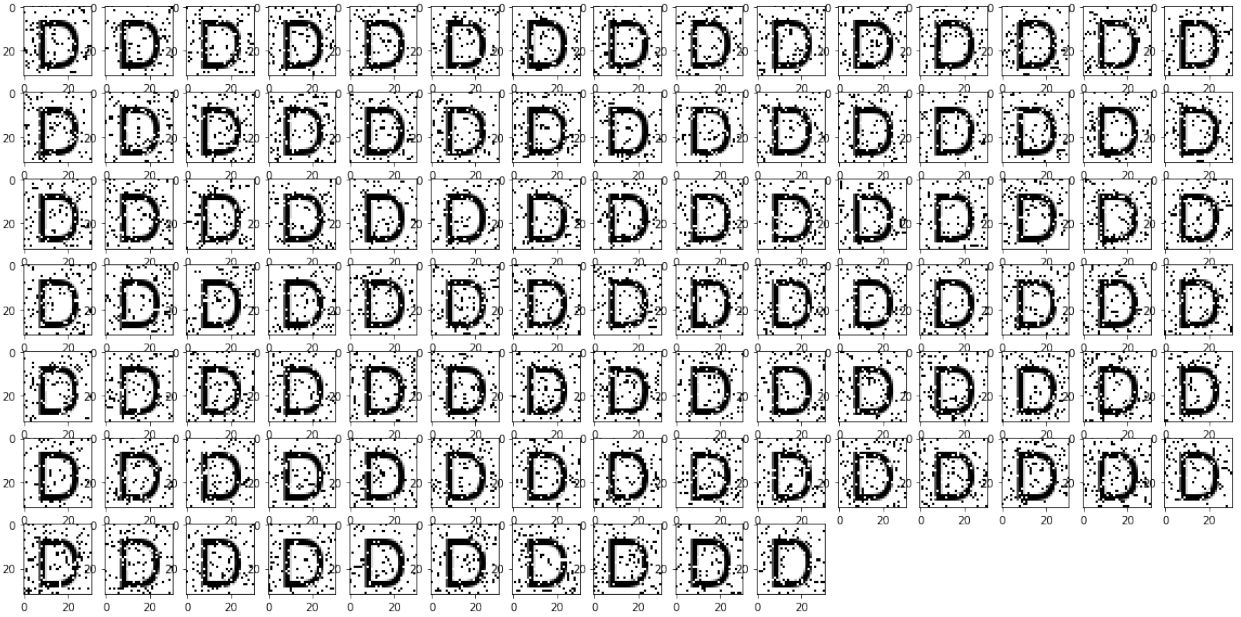
```
Training for label B...
```



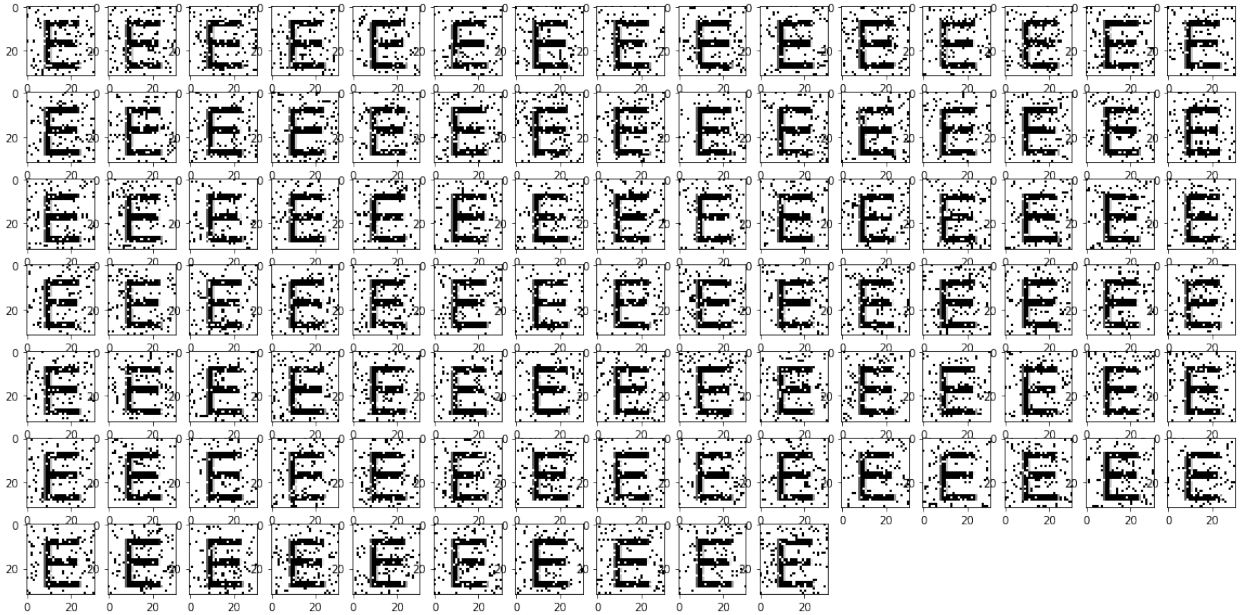
```
Training for label C...
```



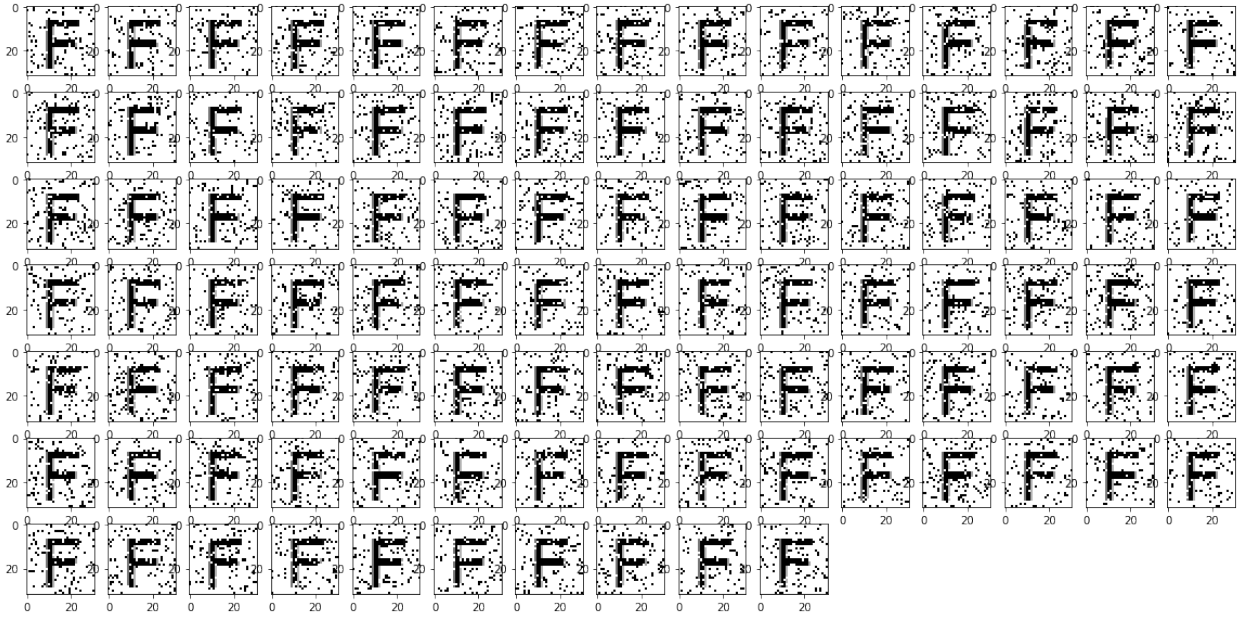
```
Training for label D...
```



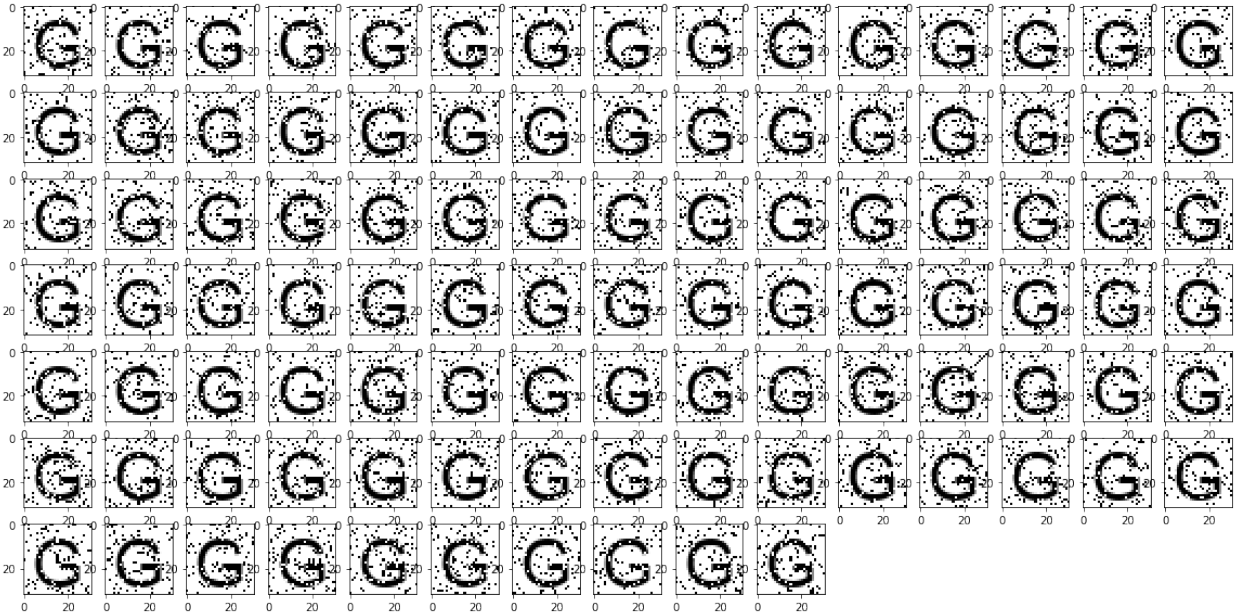
```
Training for label E...
```



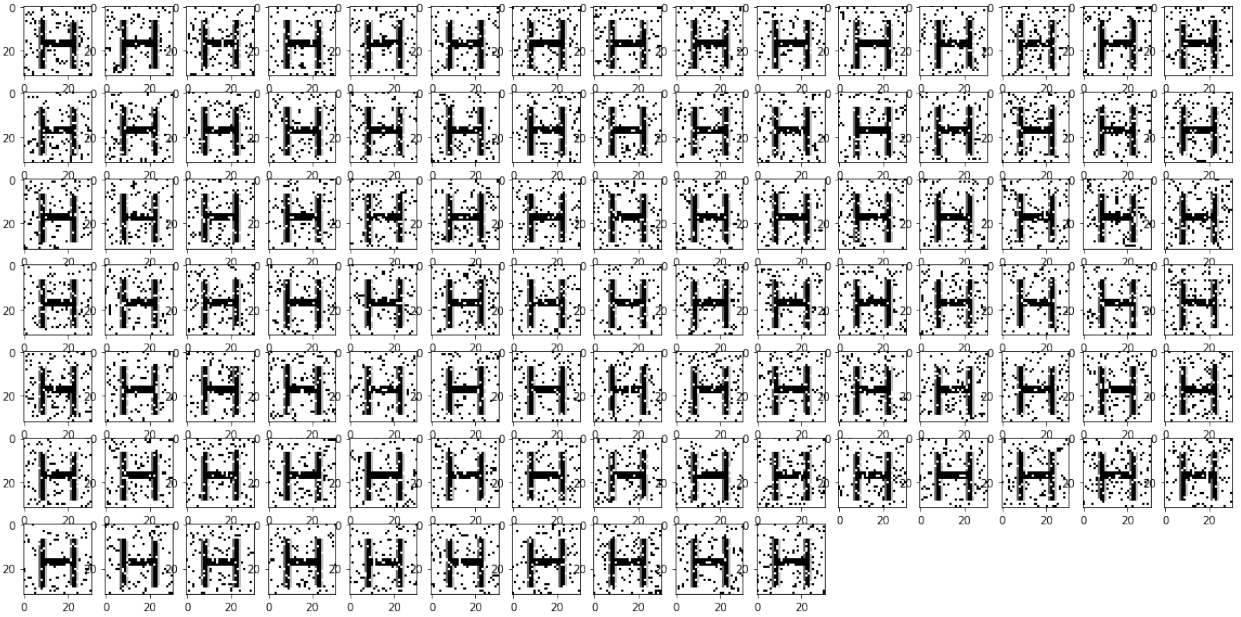
```
Training for label F...
```



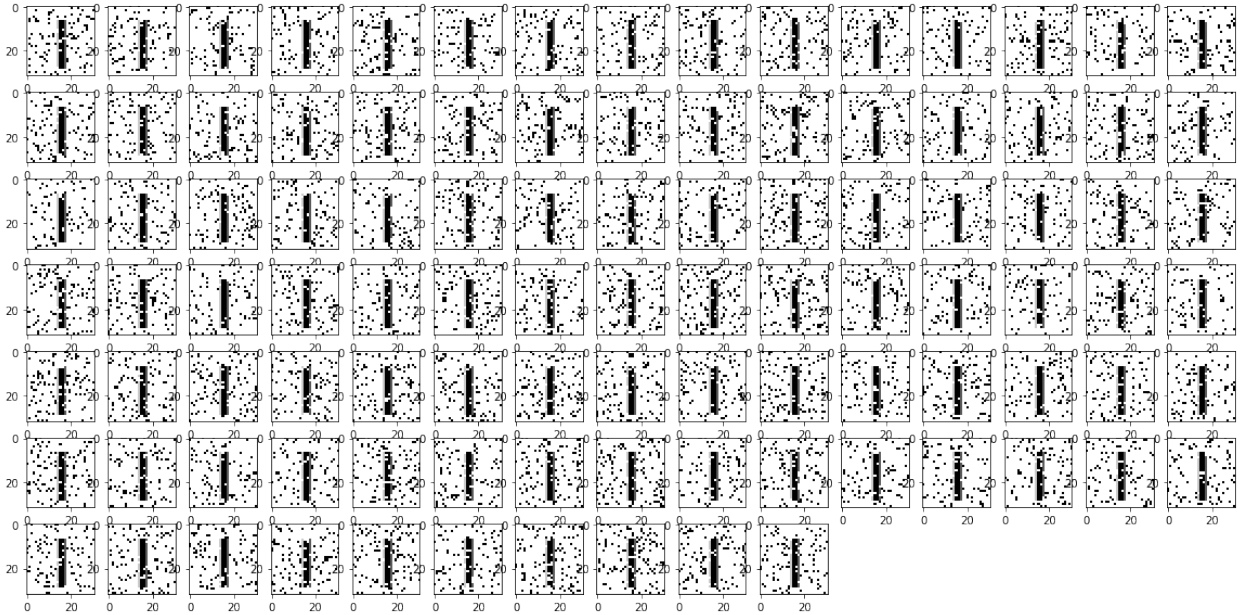
```
Training for label G...
```



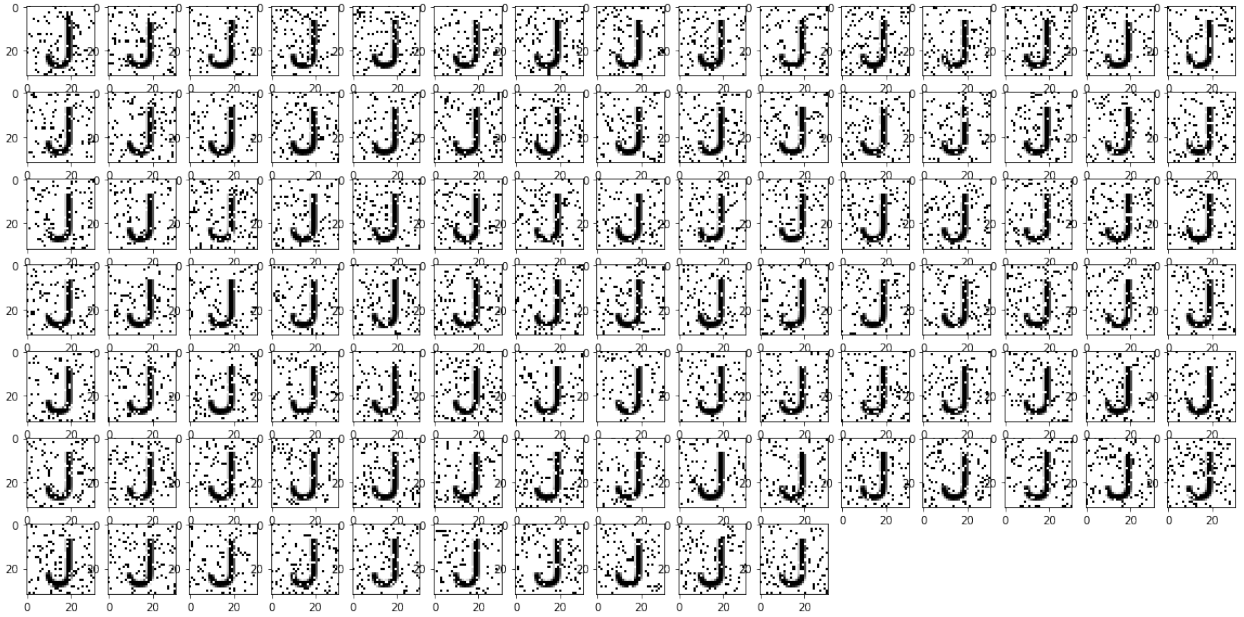
```
Training for label H...
```



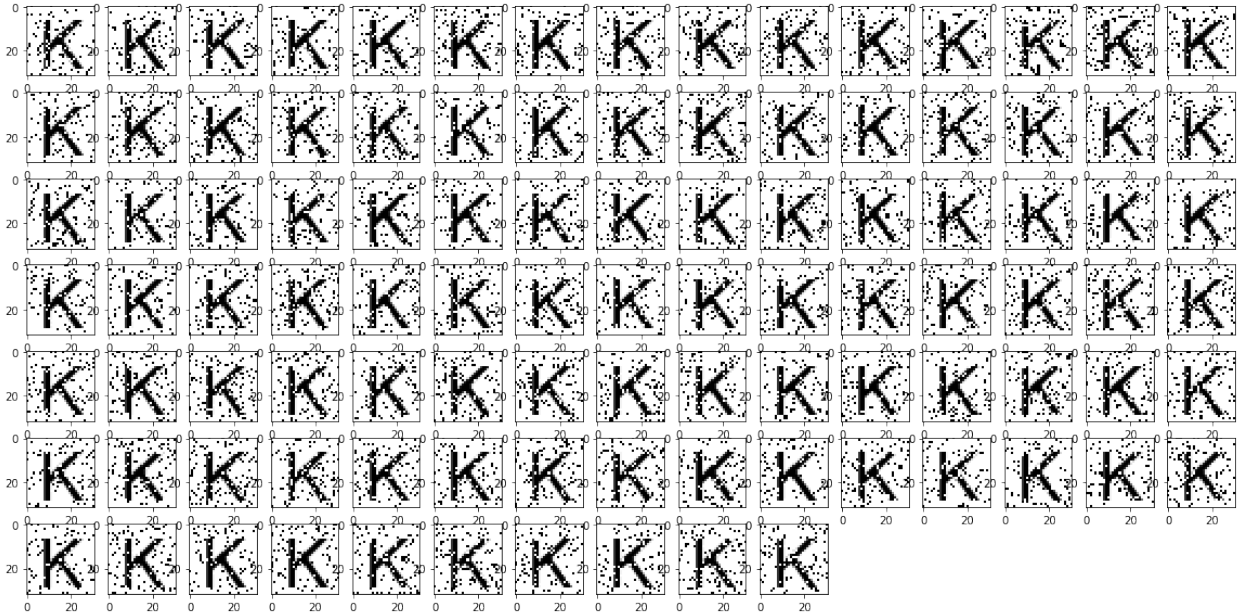
```
Training for label I...
```



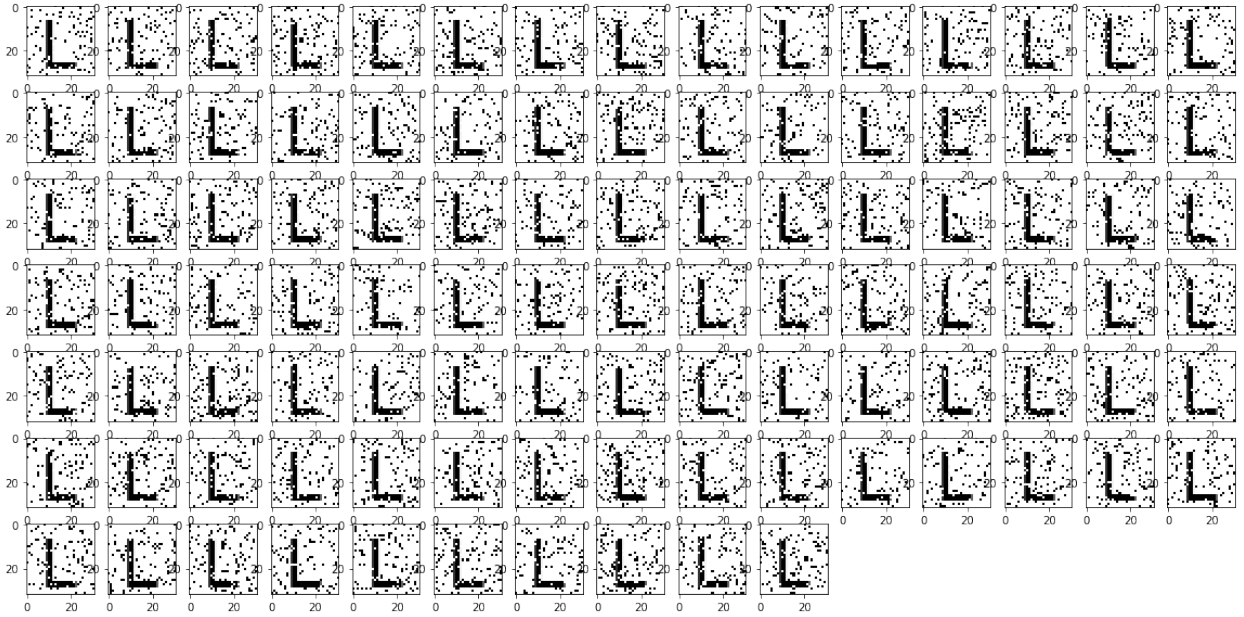
```
Training for label J...
```



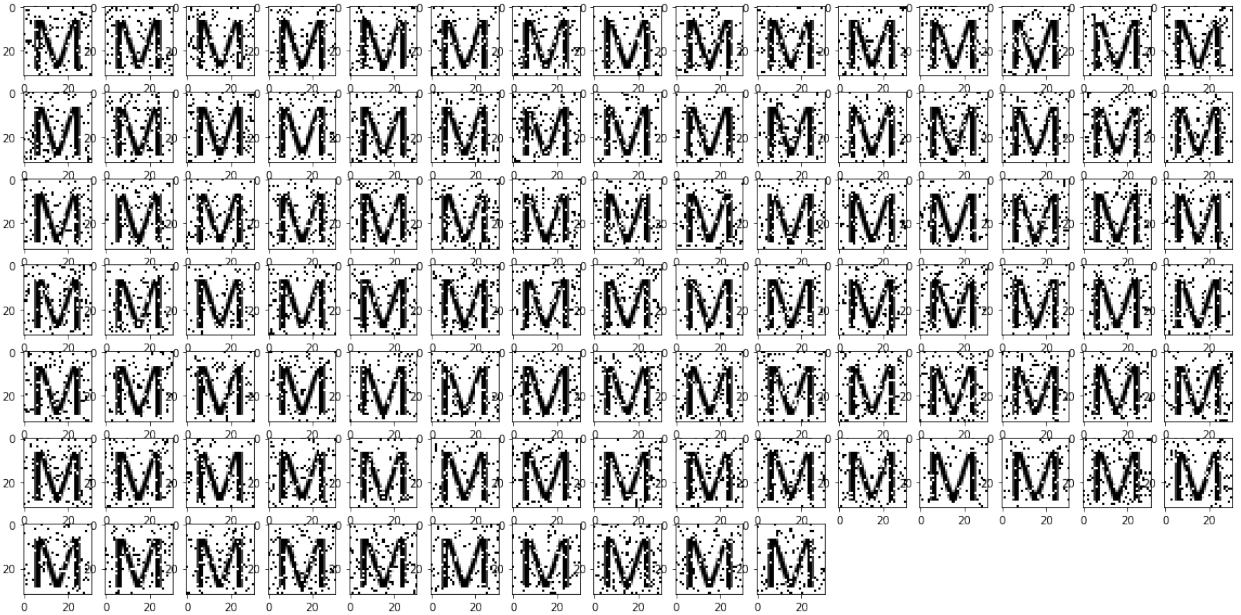
```
Training for label K...
```



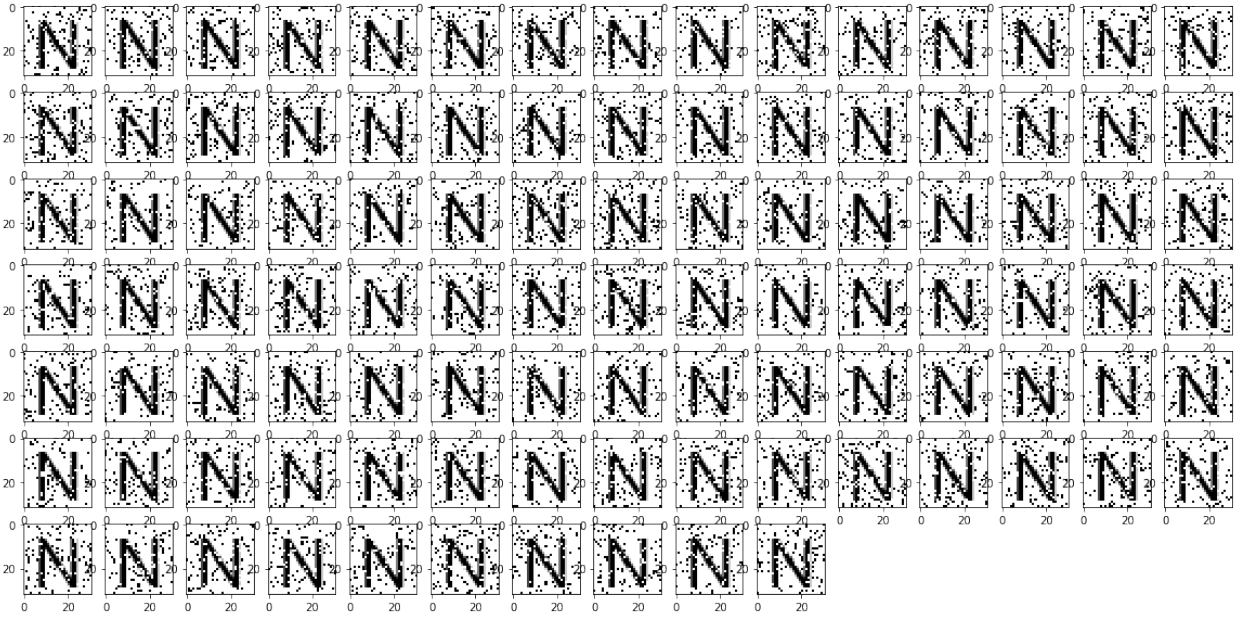
```
Training for label L...
```



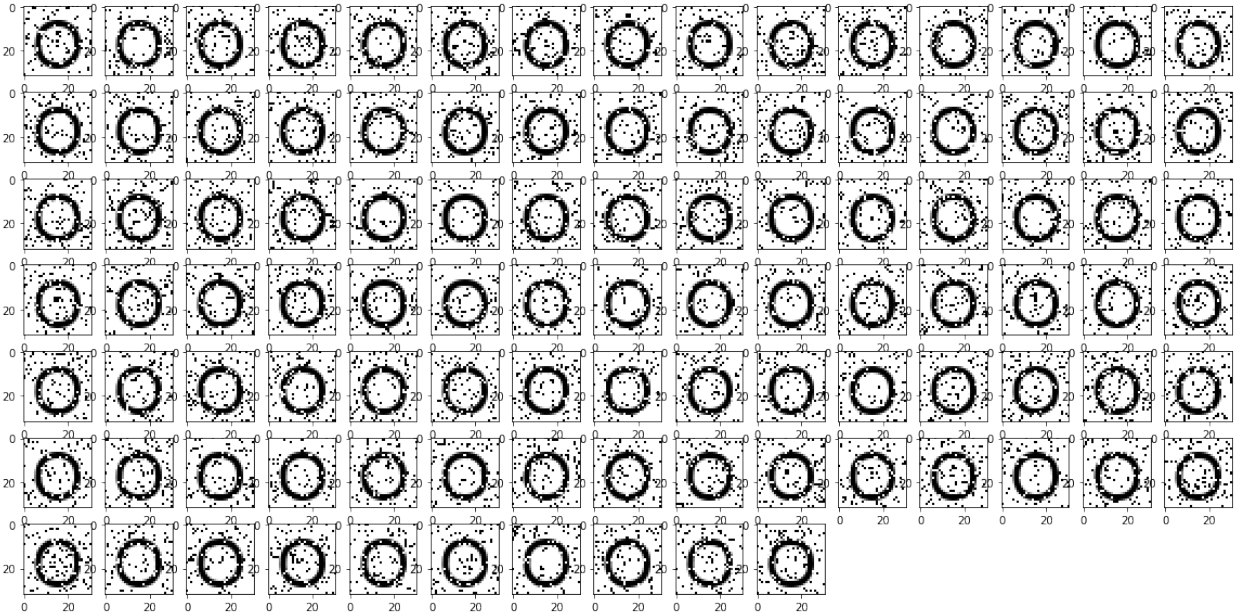
```
Training for label M...
```



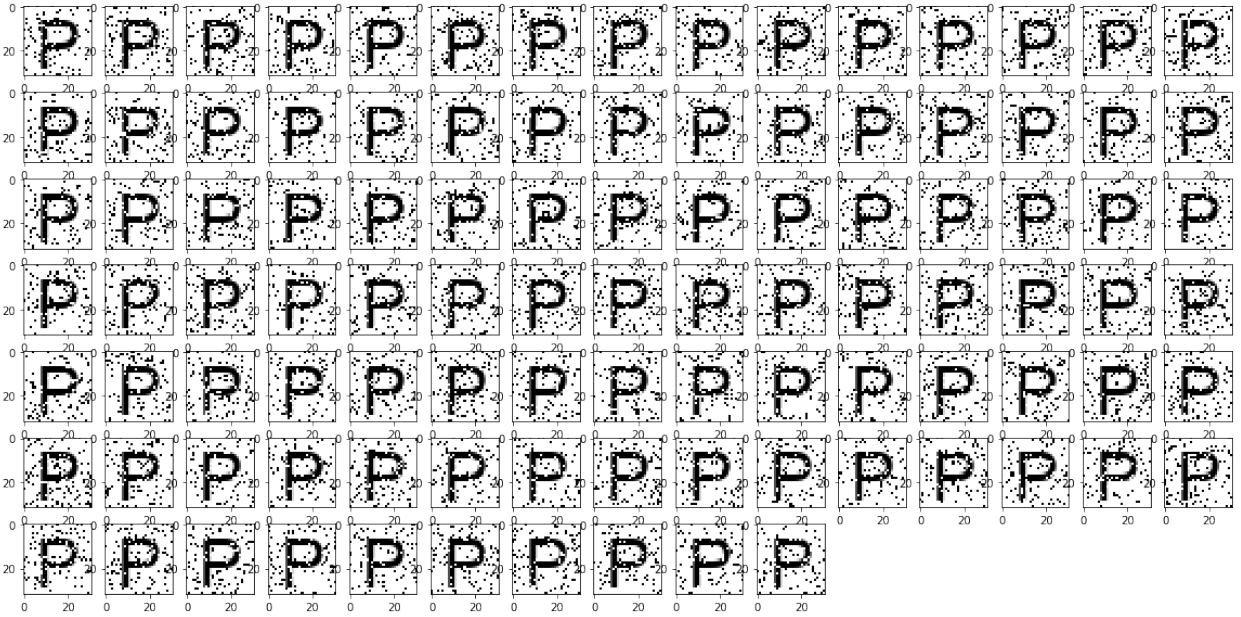
```
Training for label N...
```



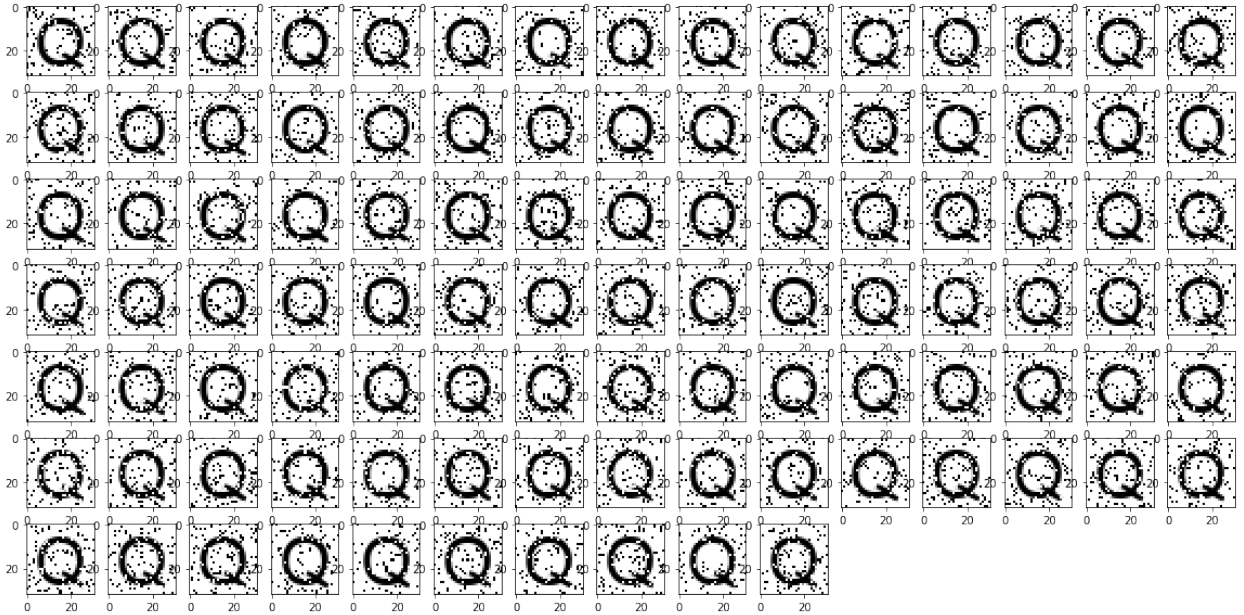
```
Training for label O...
```



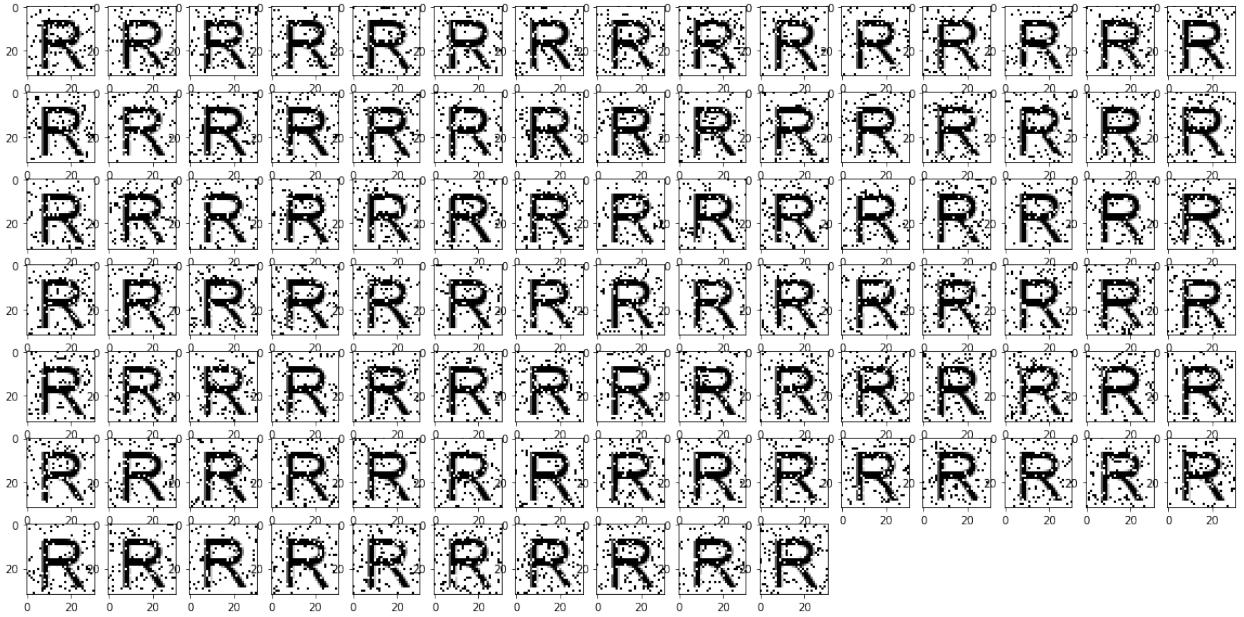
```
Training for label P...
```



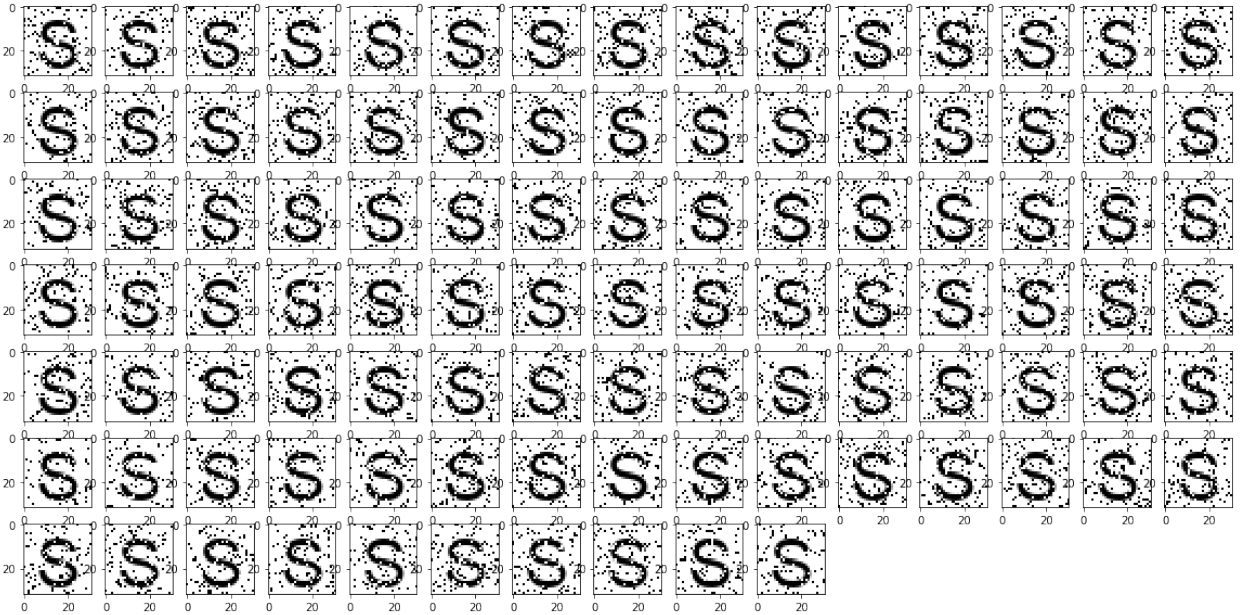
```
Training for label Q...
```



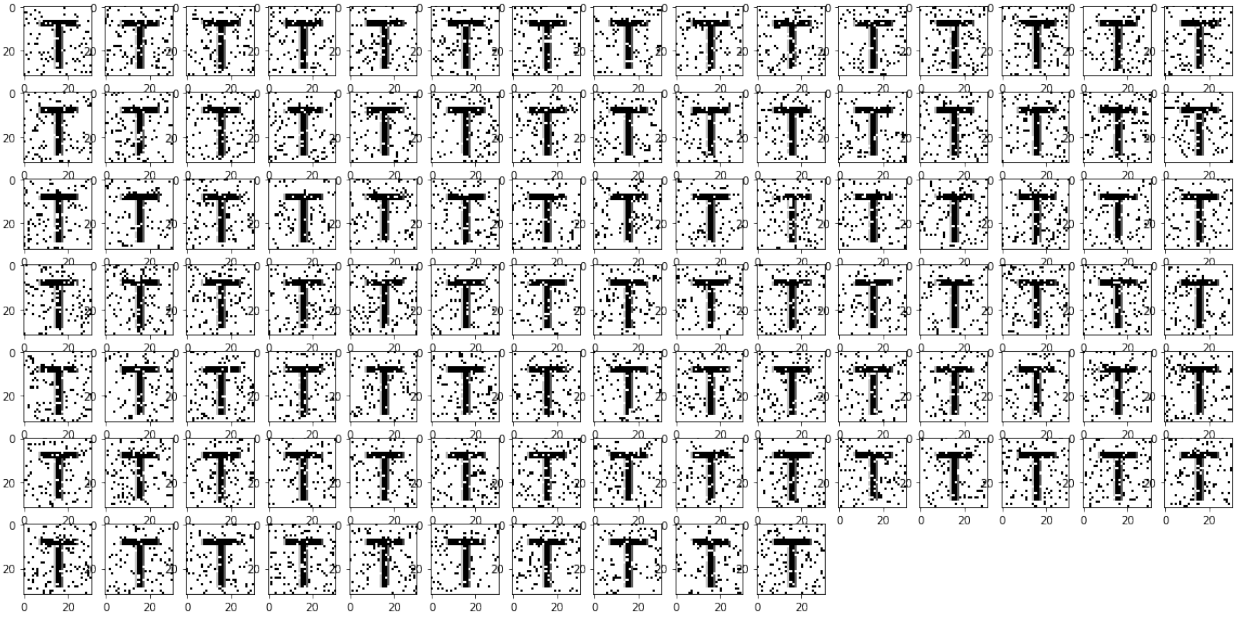
```
Training for label R...
```



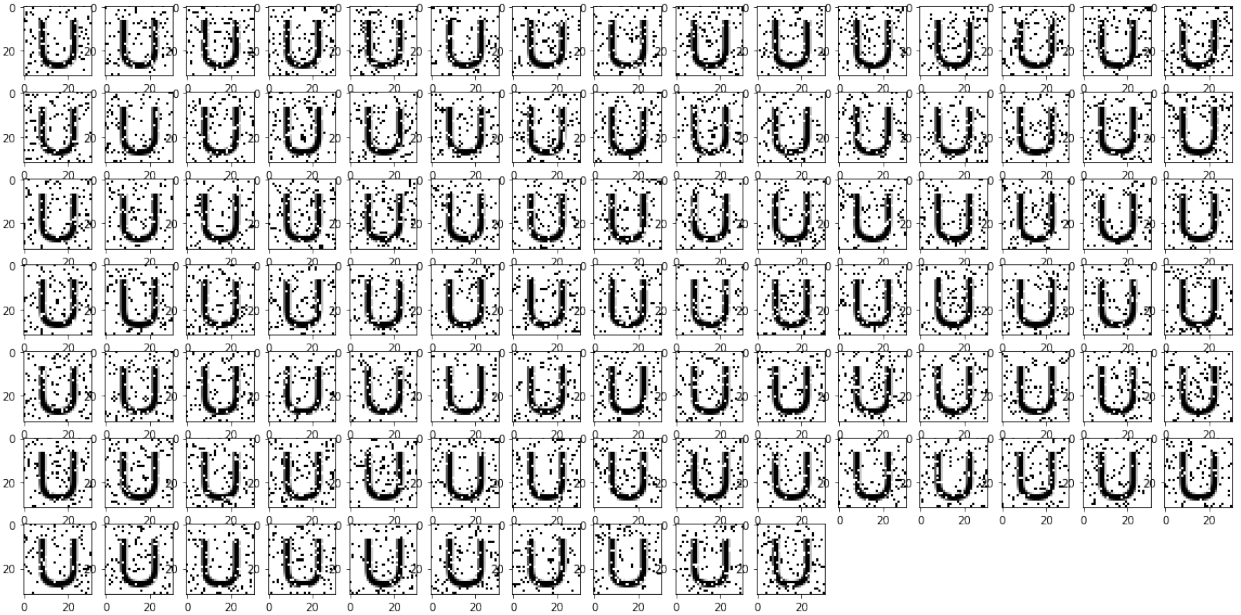
```
Training for label S...
```



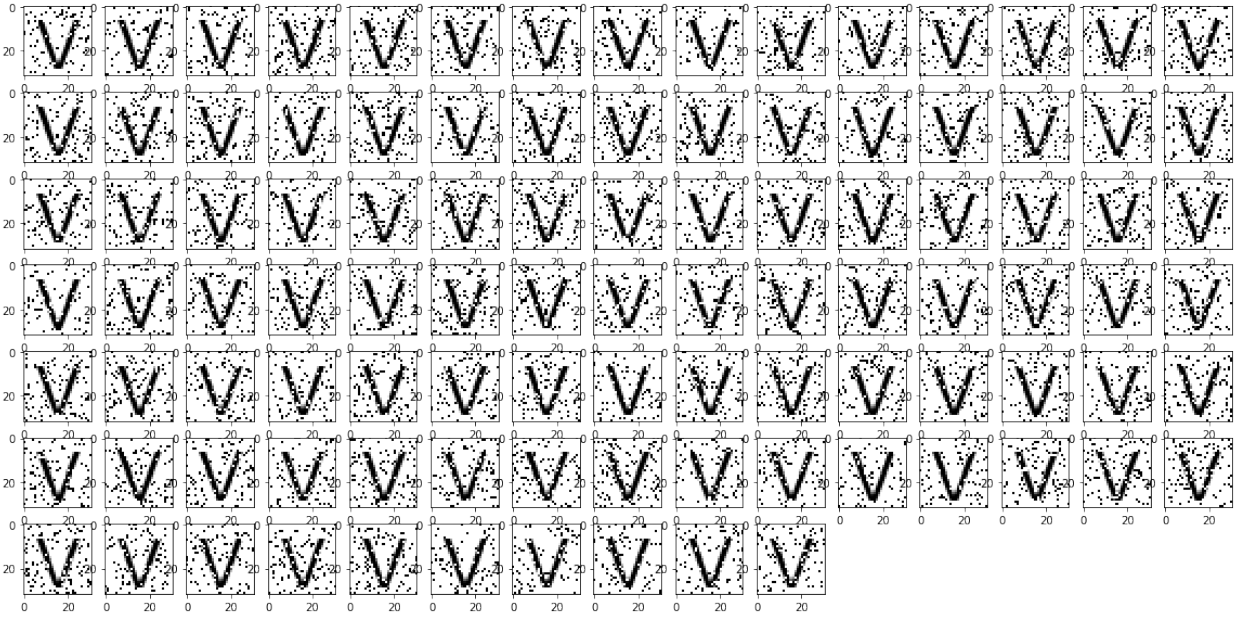
```
Training for label T...
```



```
Training for label U...
```

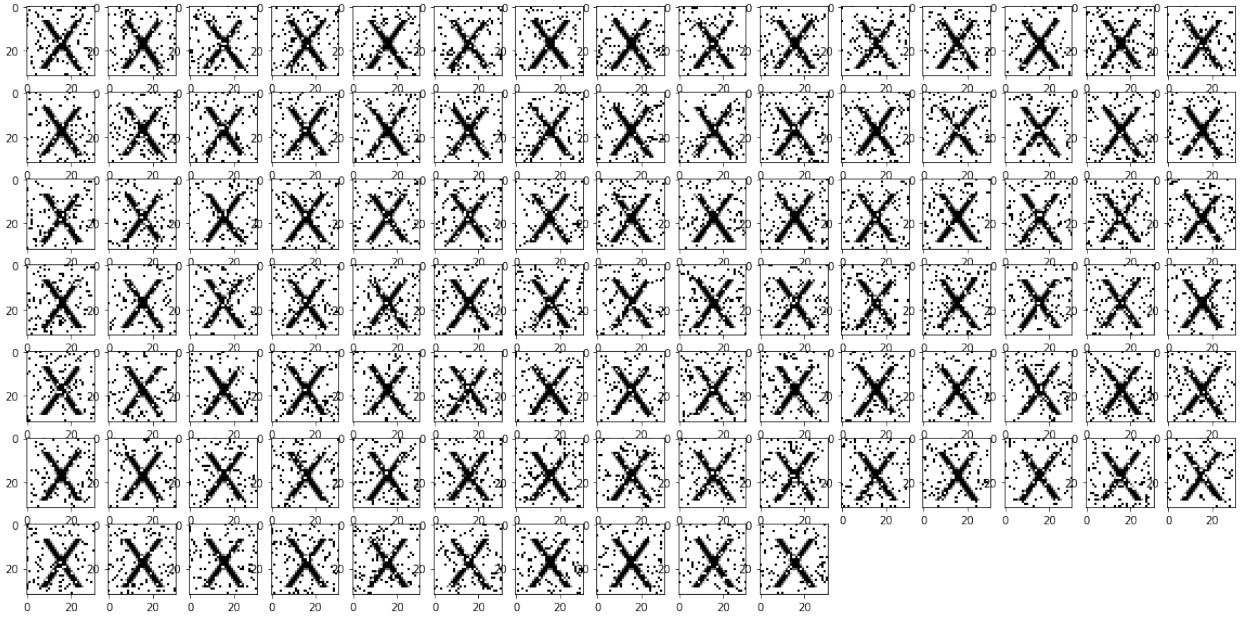


```
Training for label V...
```

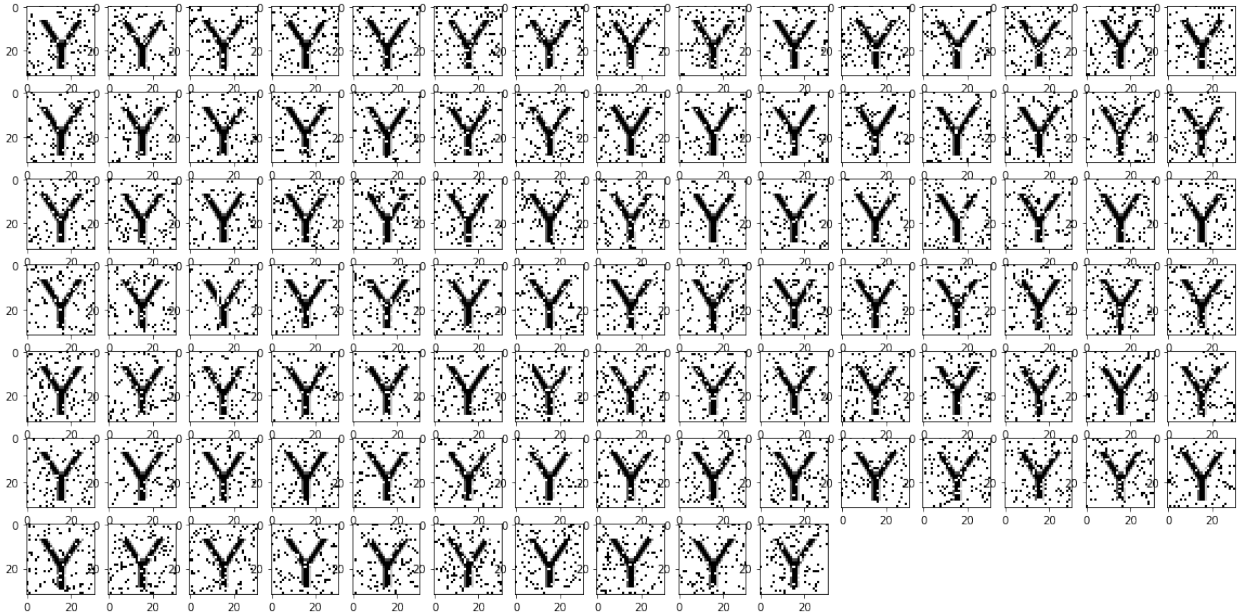


```
Training for label W...
```

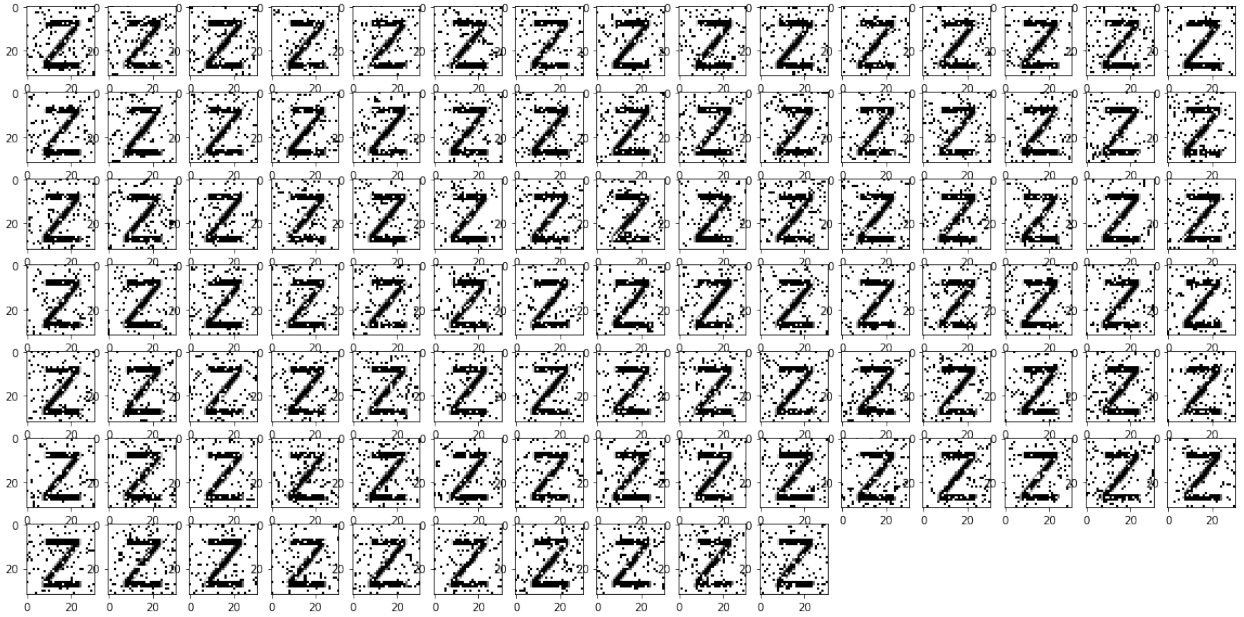
```
Training for label X...
```



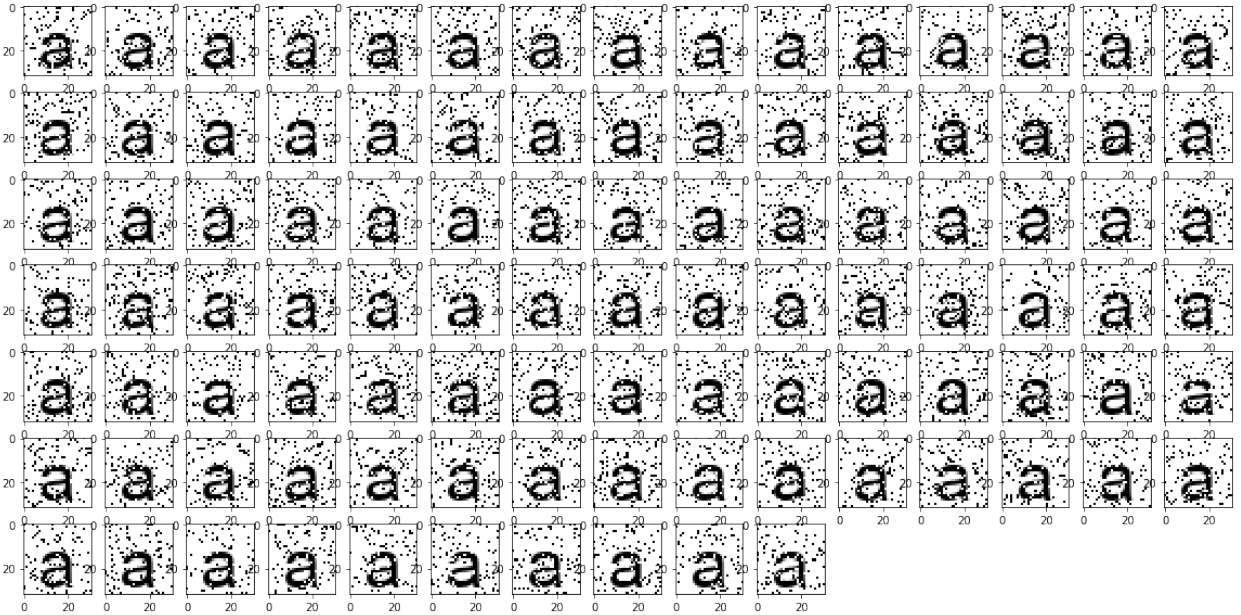
```
Training for label Y...
```



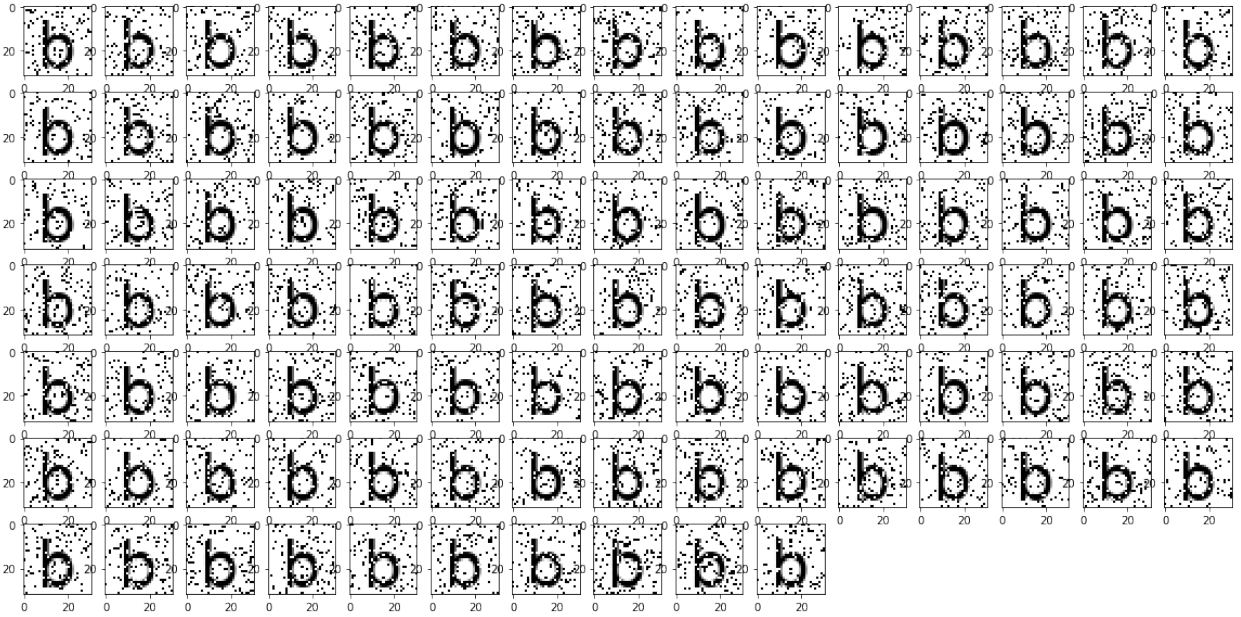
```
Training for label Z...
```



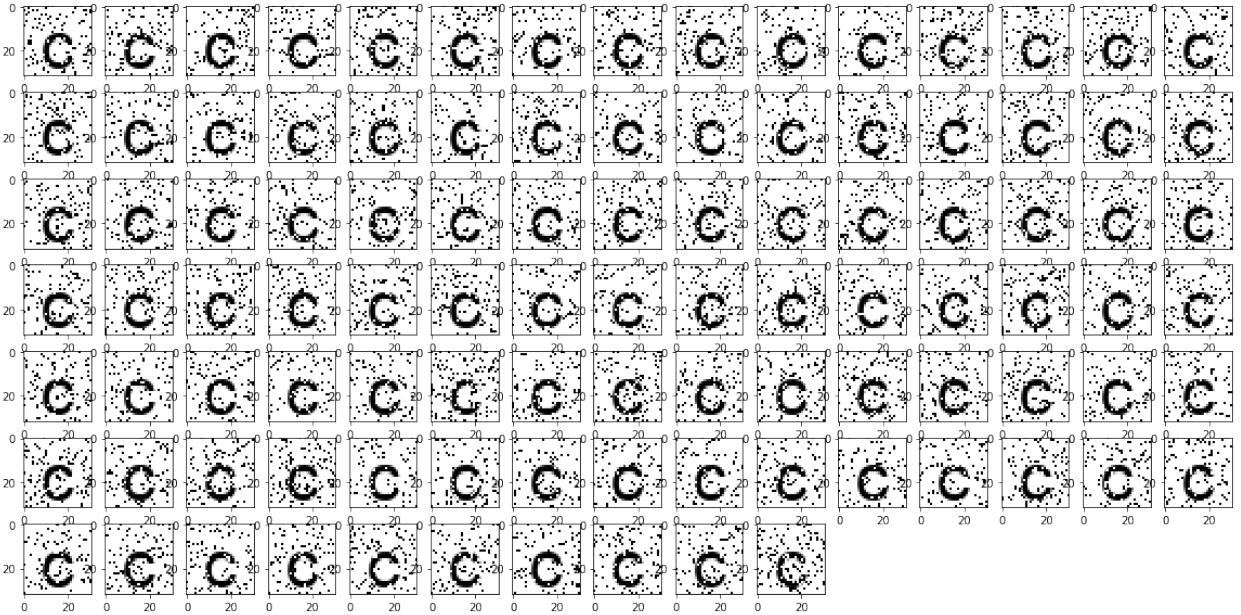
```
Training for label a...
```



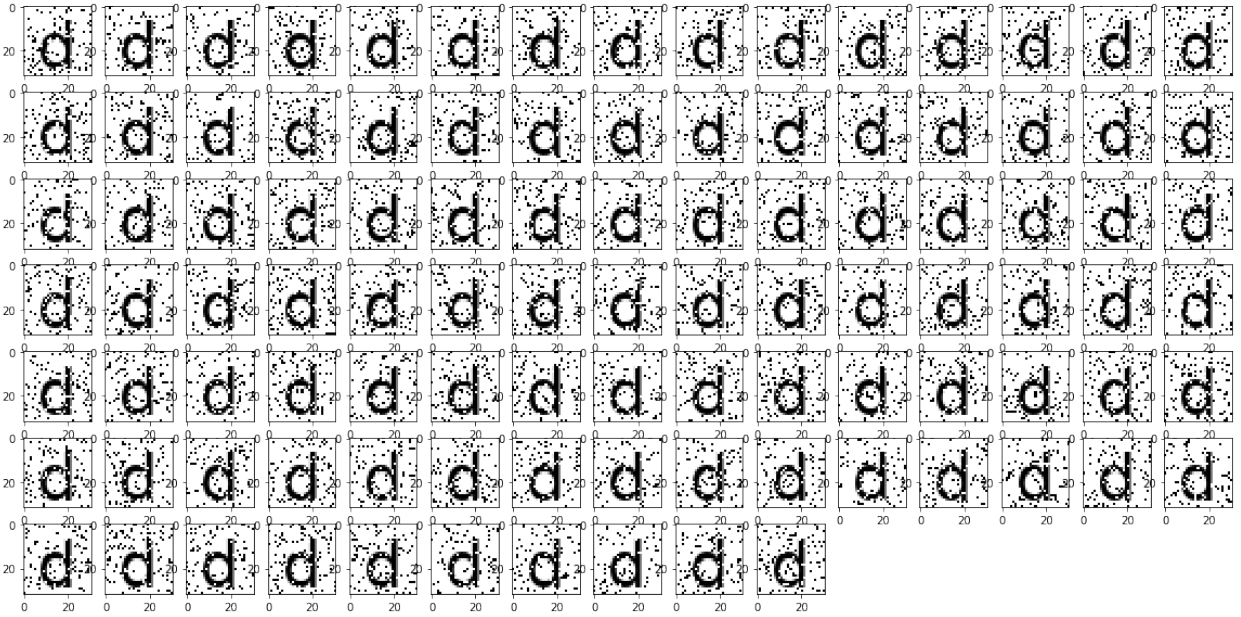
```
Training for label b...
```



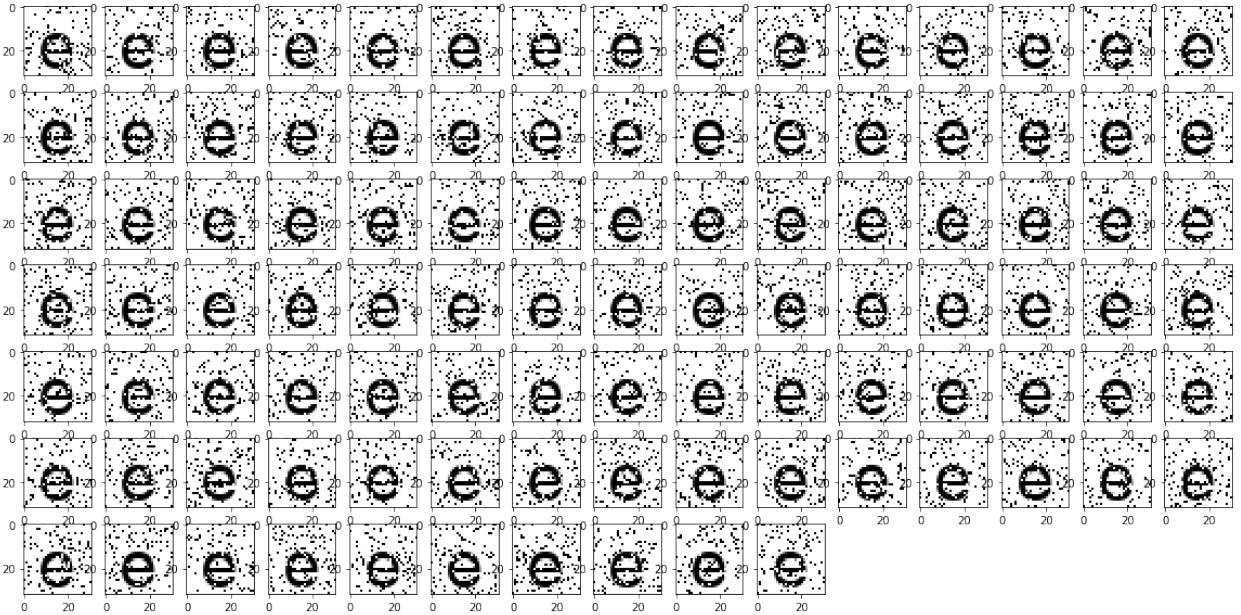
```
Training for label c...
```



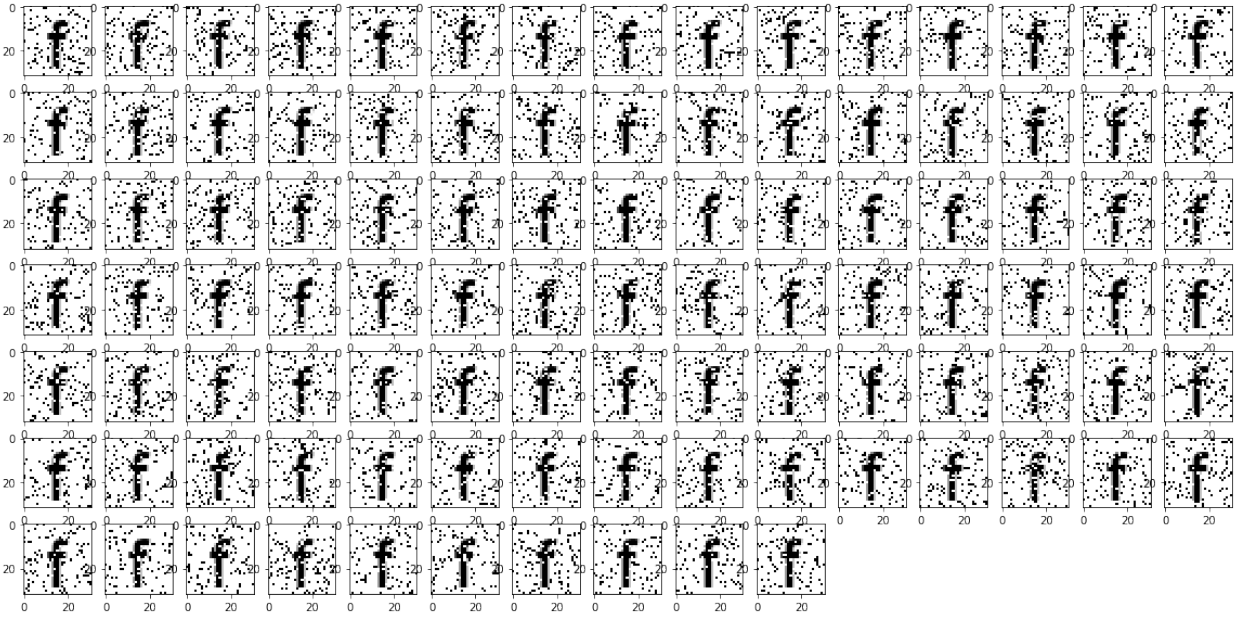
```
Training for label d...
```



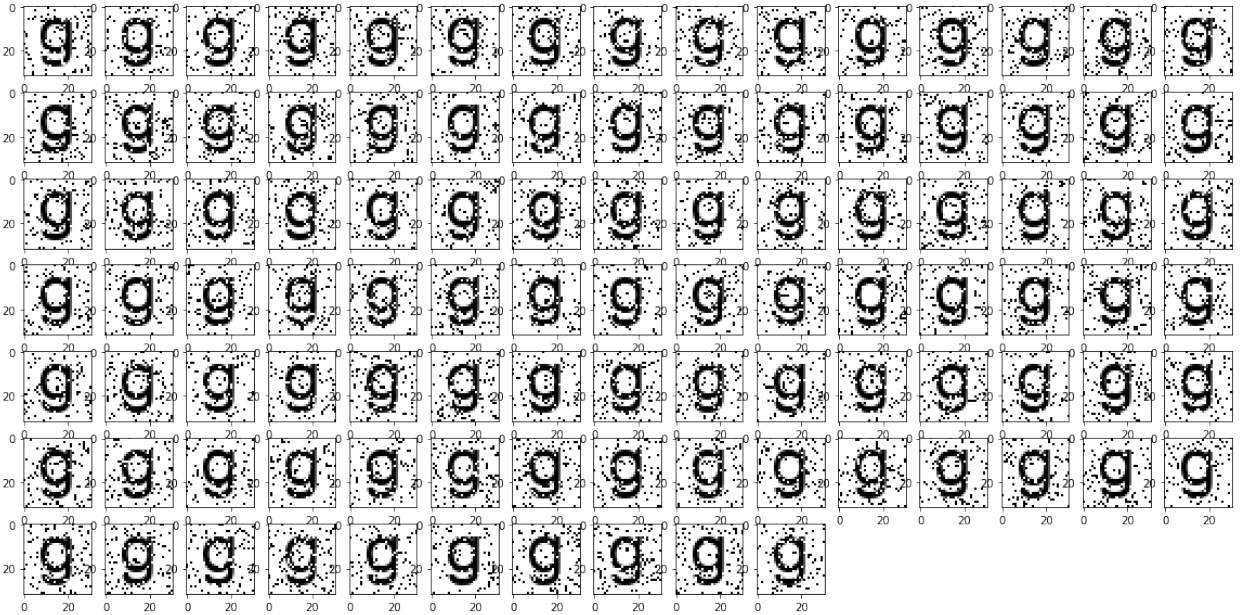
```
Training for label e...
```



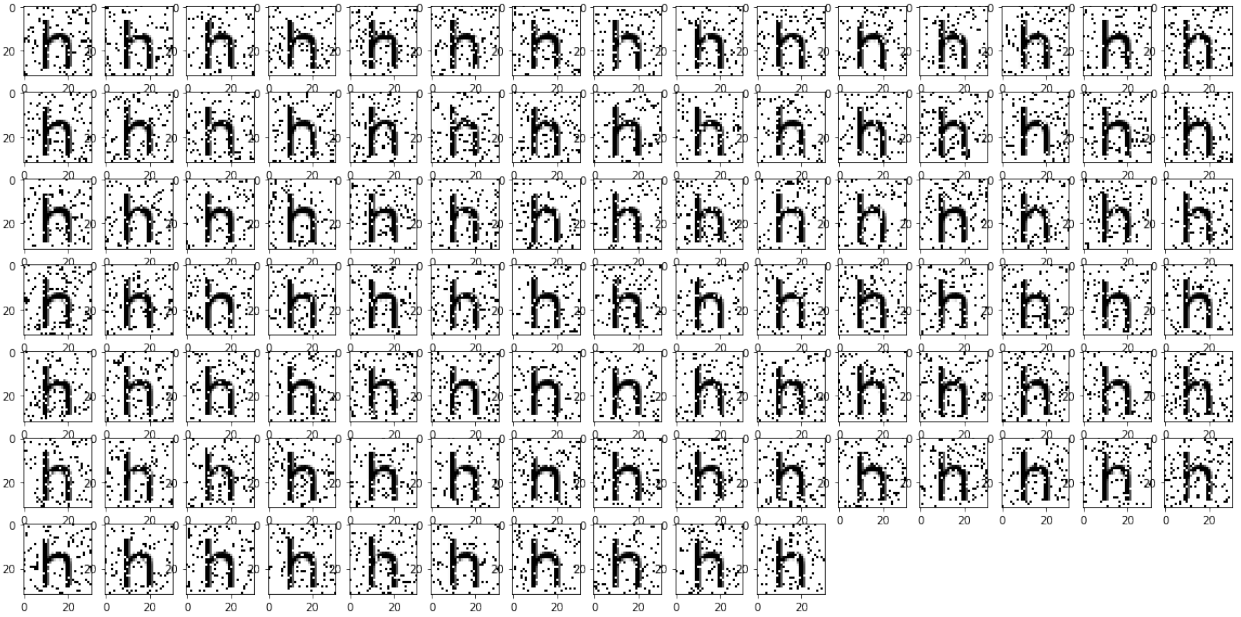
```
Training for label f...
```



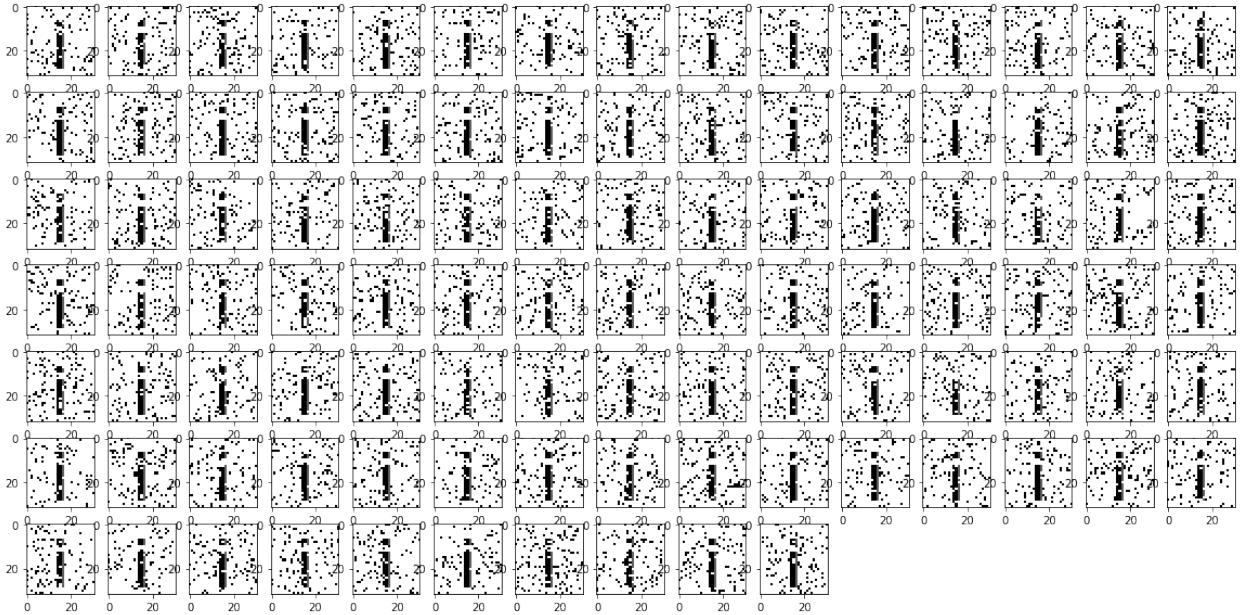
```
Training for label g...
```



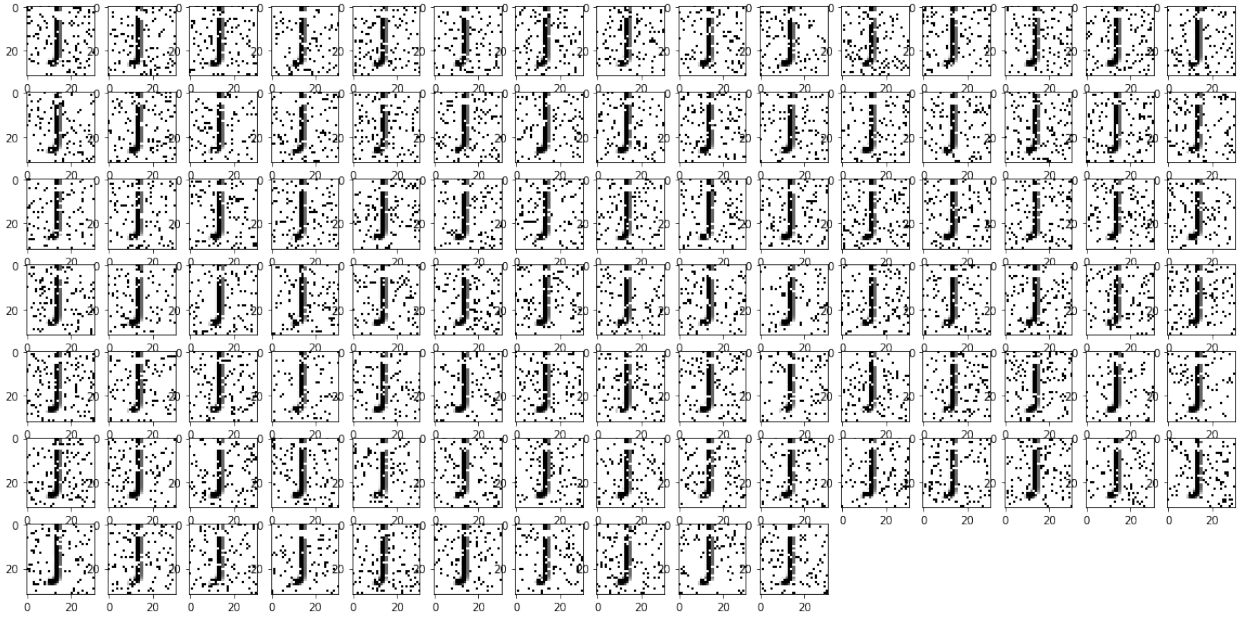
```
Training for label h...
```



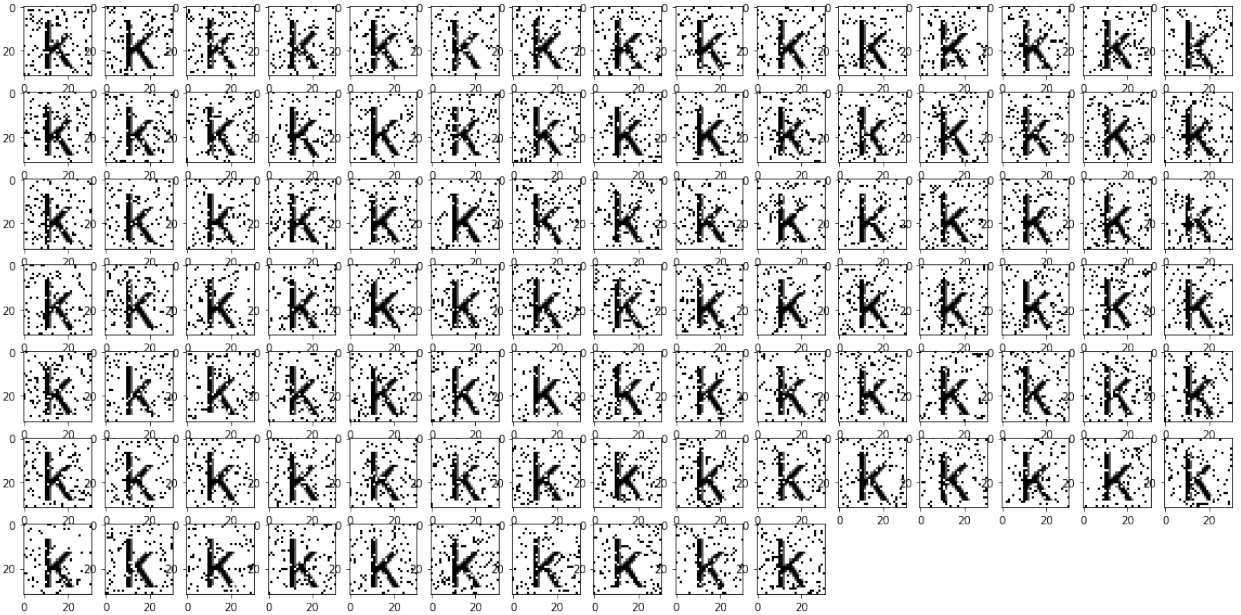
```
Training for label i...
```



```
Training for label j...
```

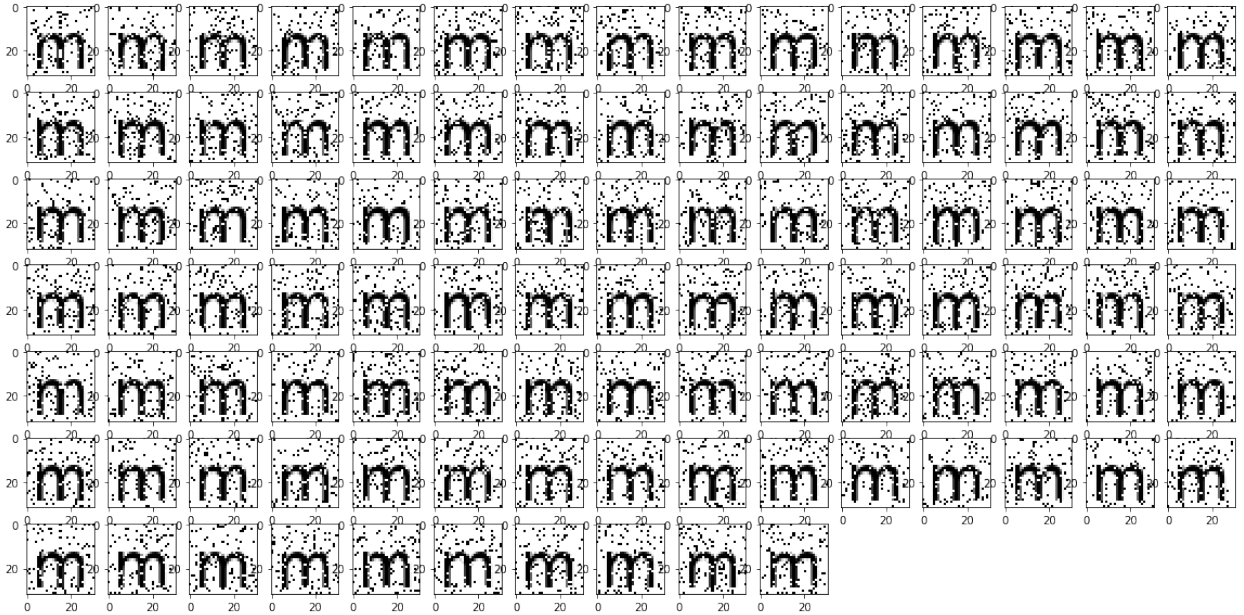


```
Training for label k...
```

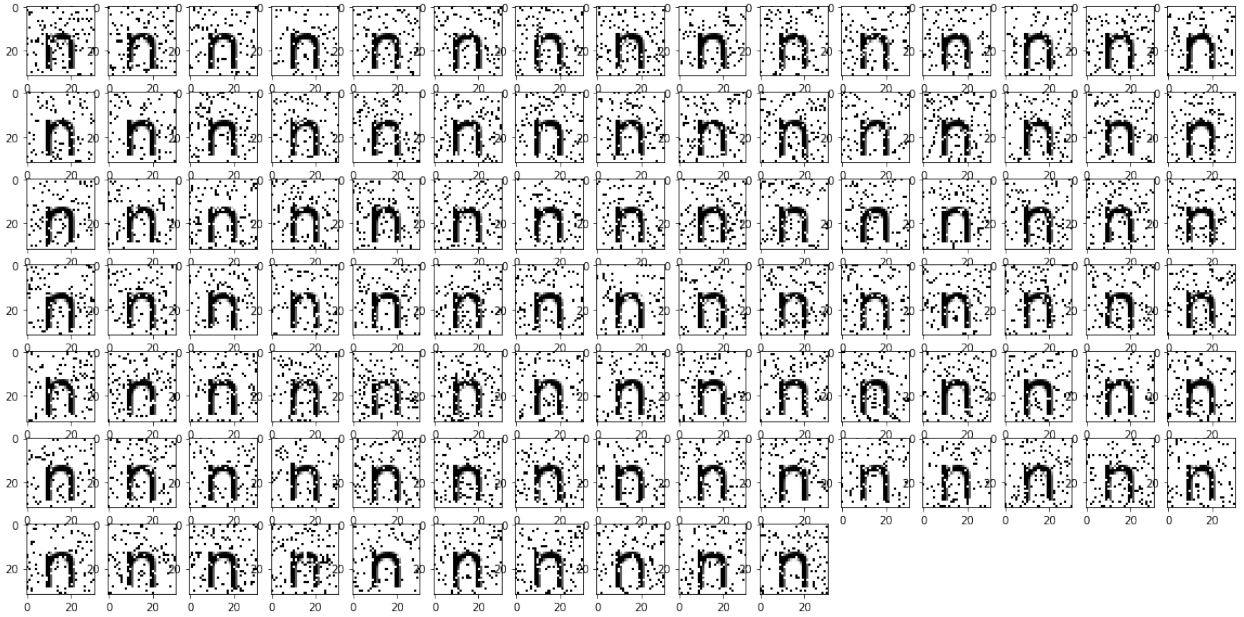


```
Training for label l...
```

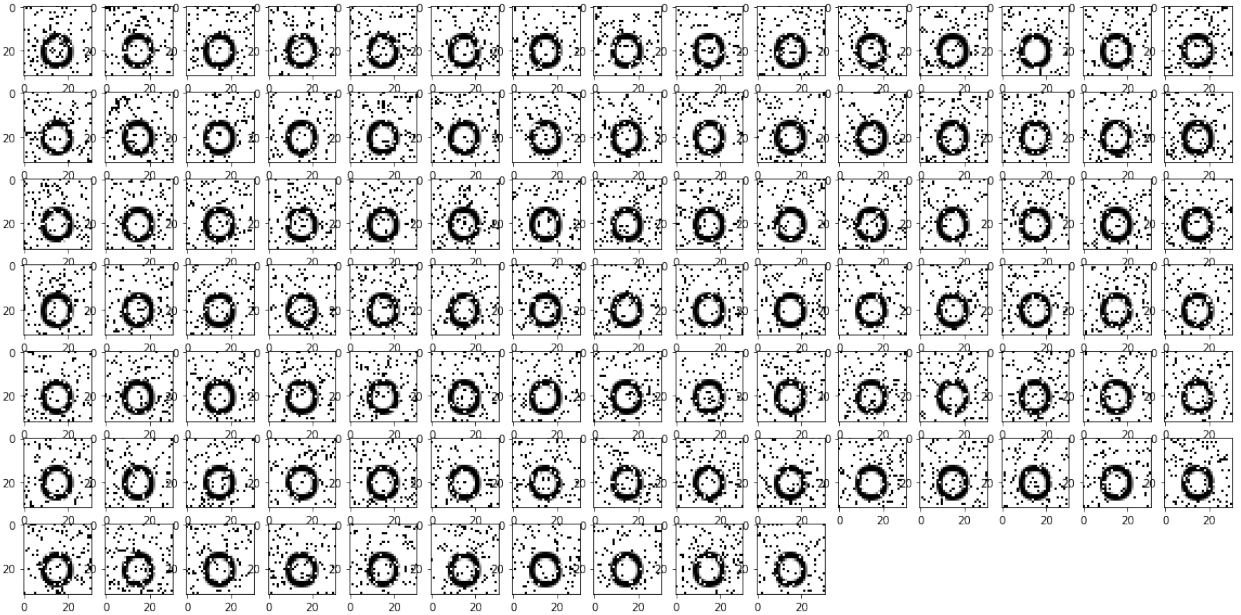
```
Training for label m...
```



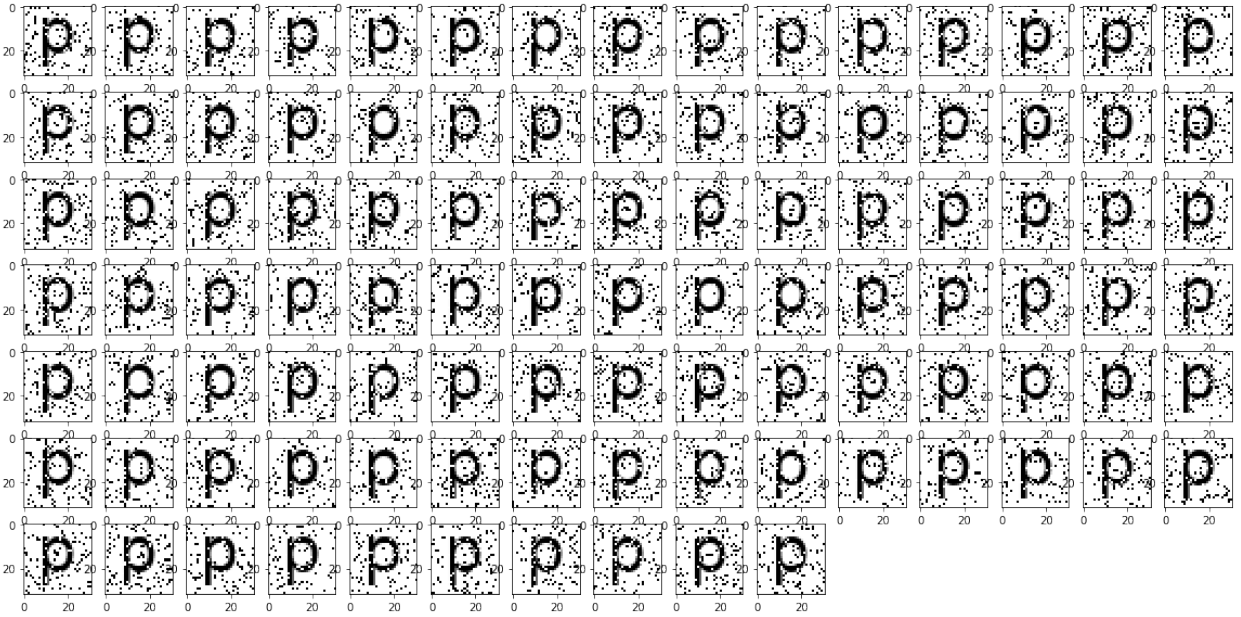
```
Training for label n...
```



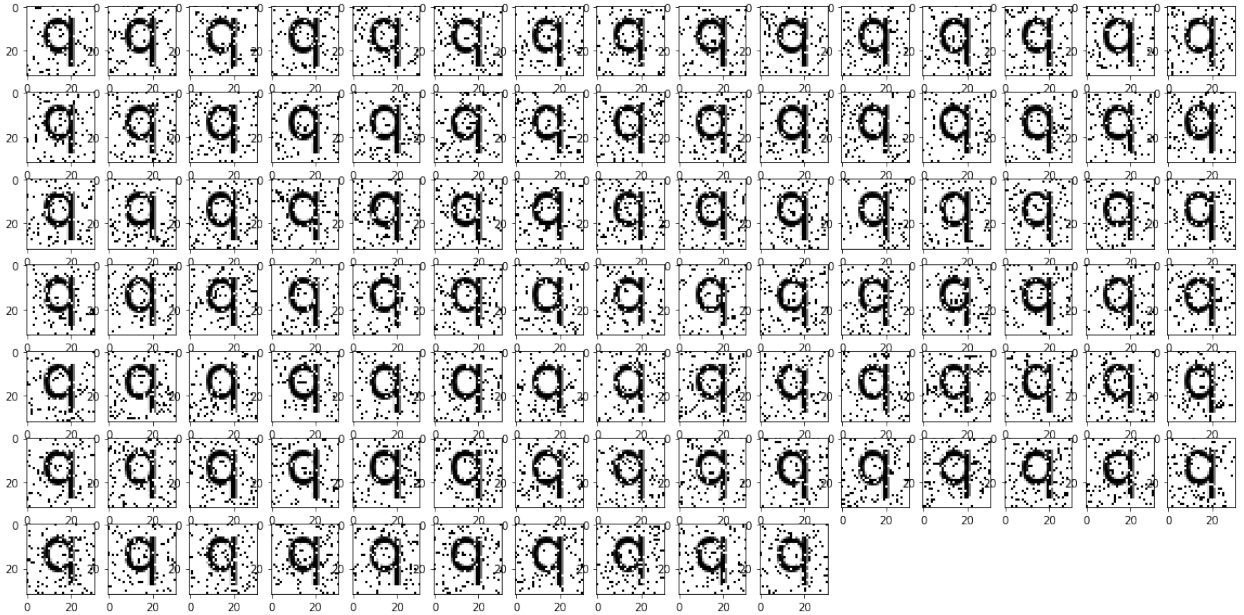
```
Training for label o...
```



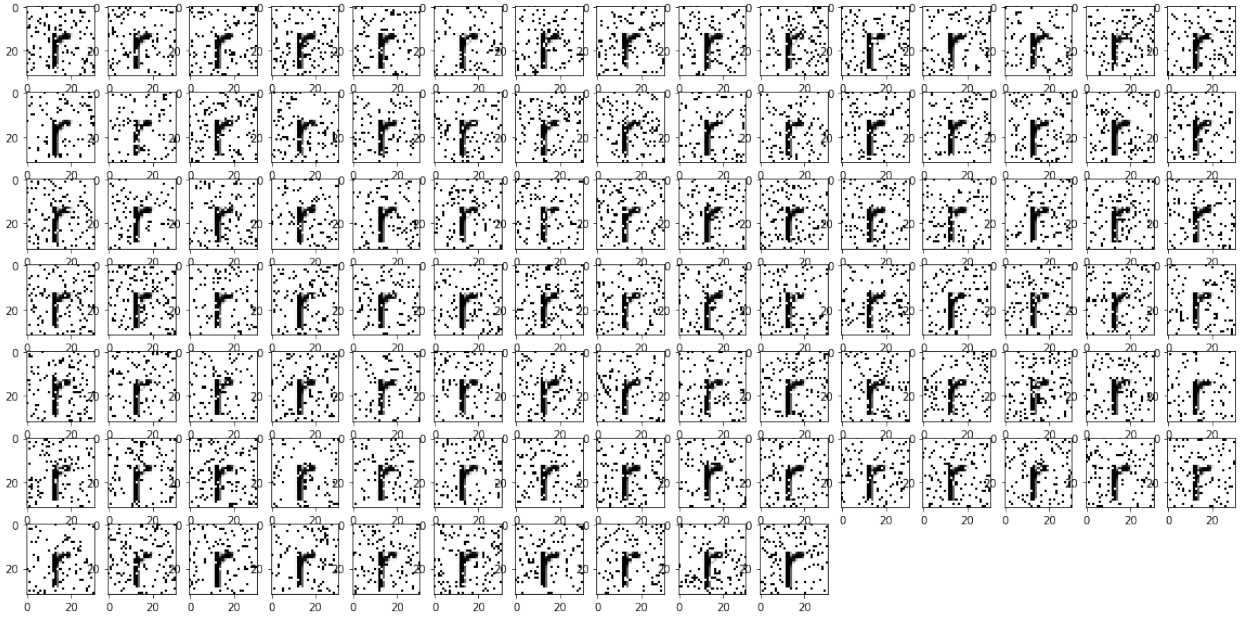
```
Training for label p...
```



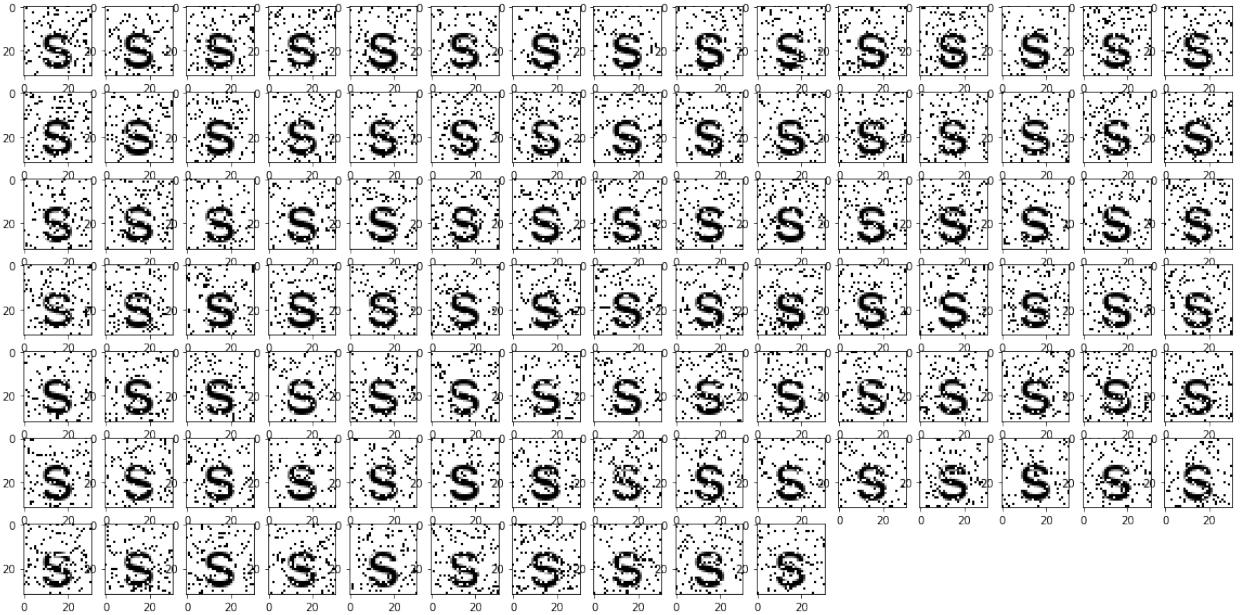
```
Training for label q...
```



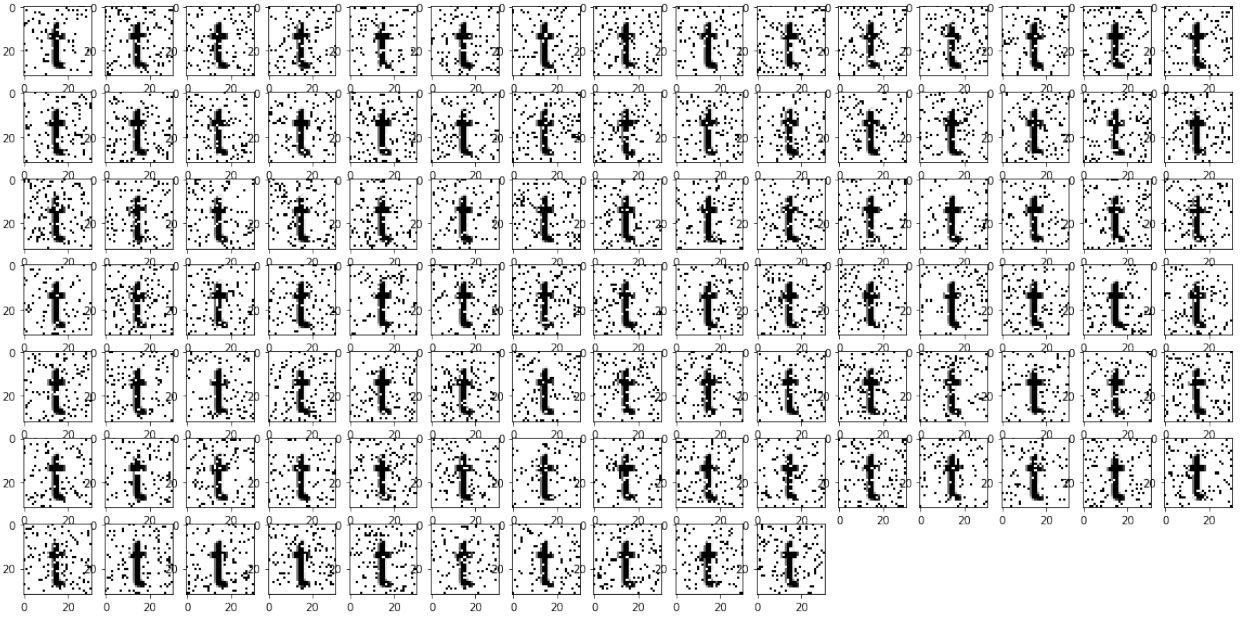
```
Training for label r...
```



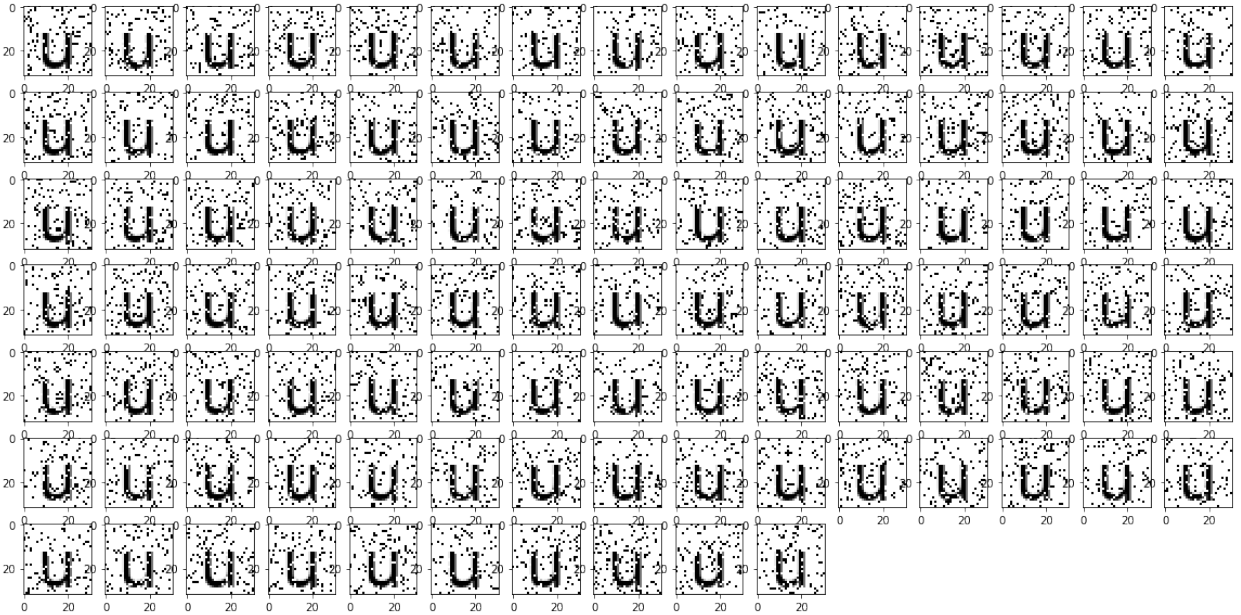
```
Training for label s...
```



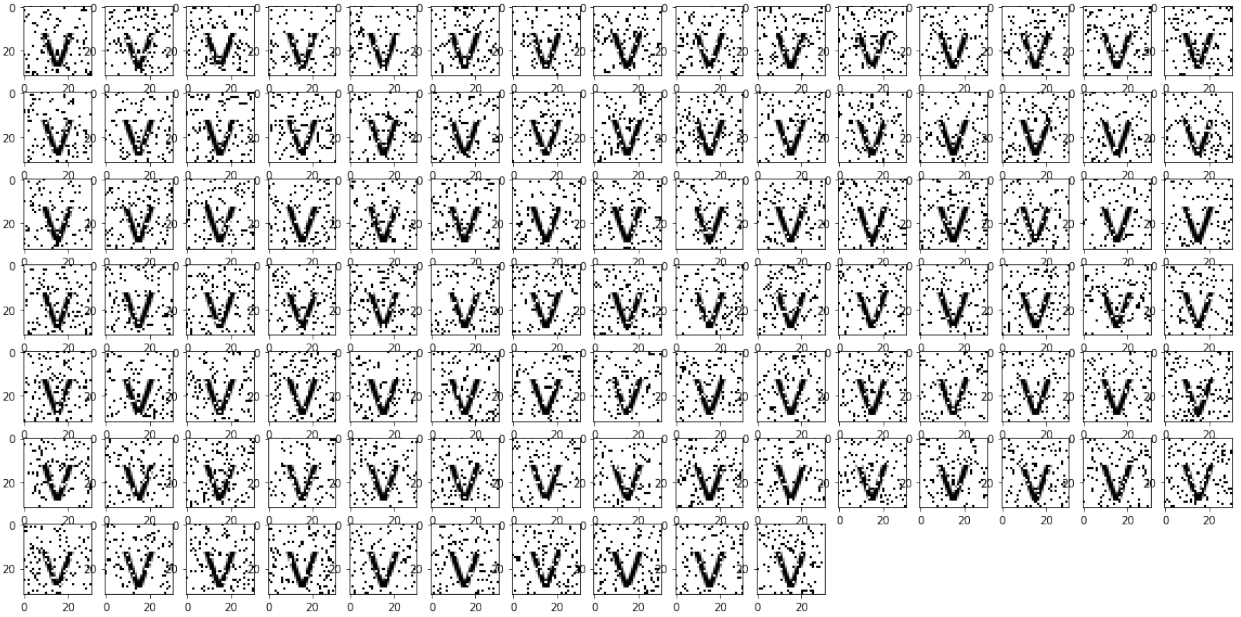
```
Training for label t...
```



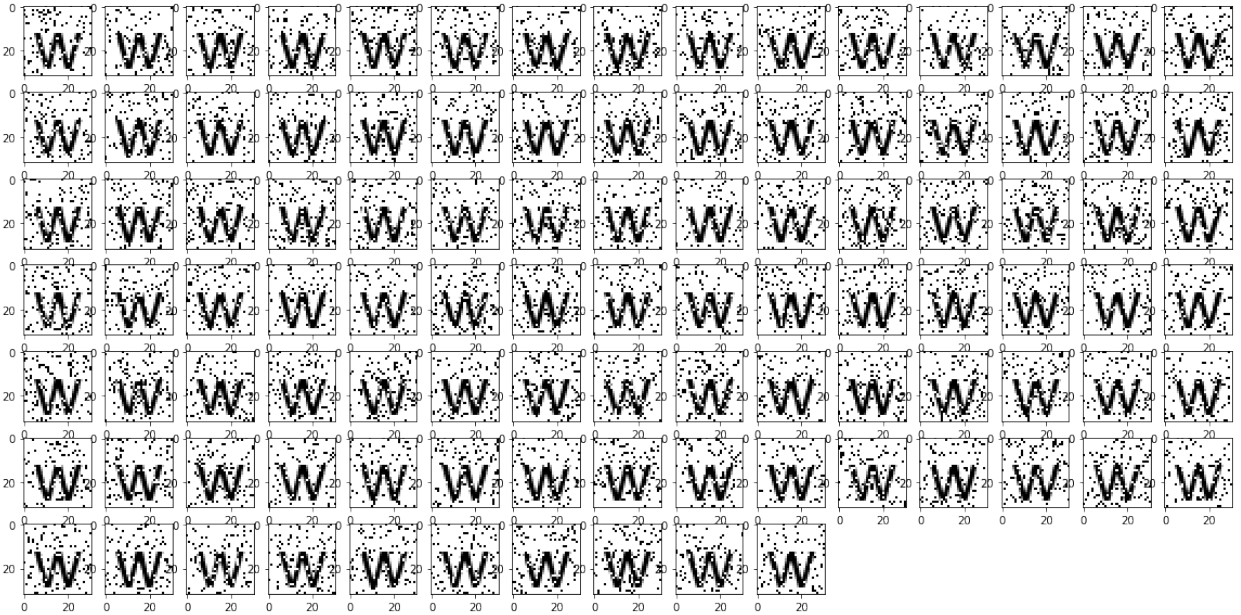
```
Training for label u...
```



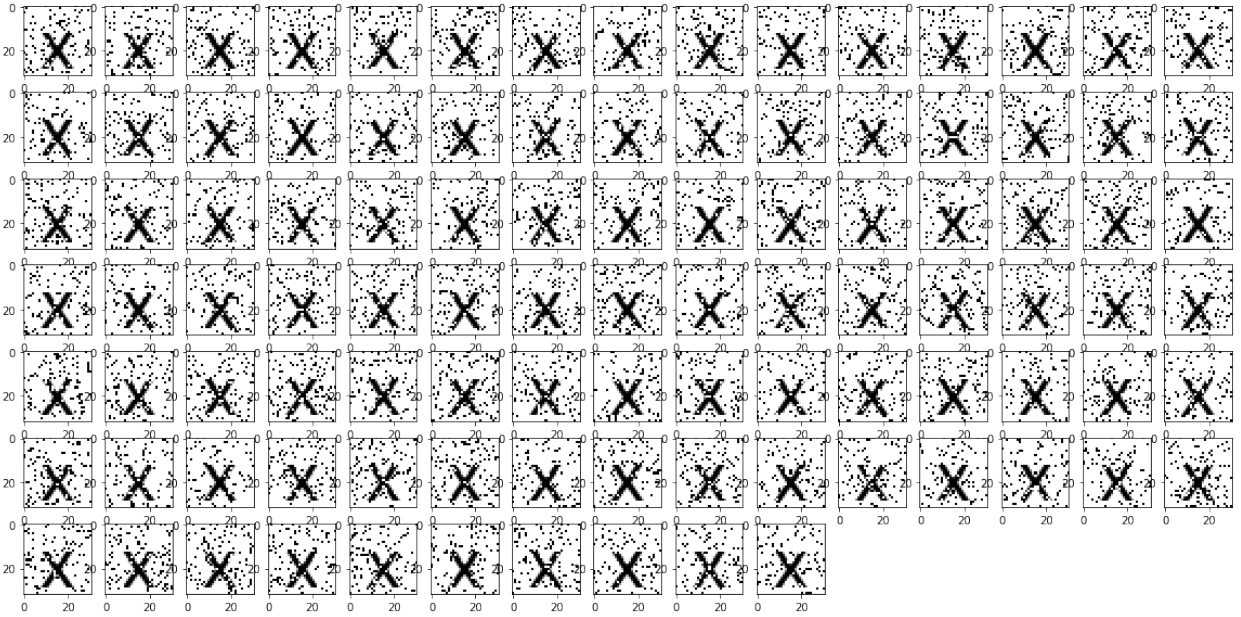
```
Training for label v...
```



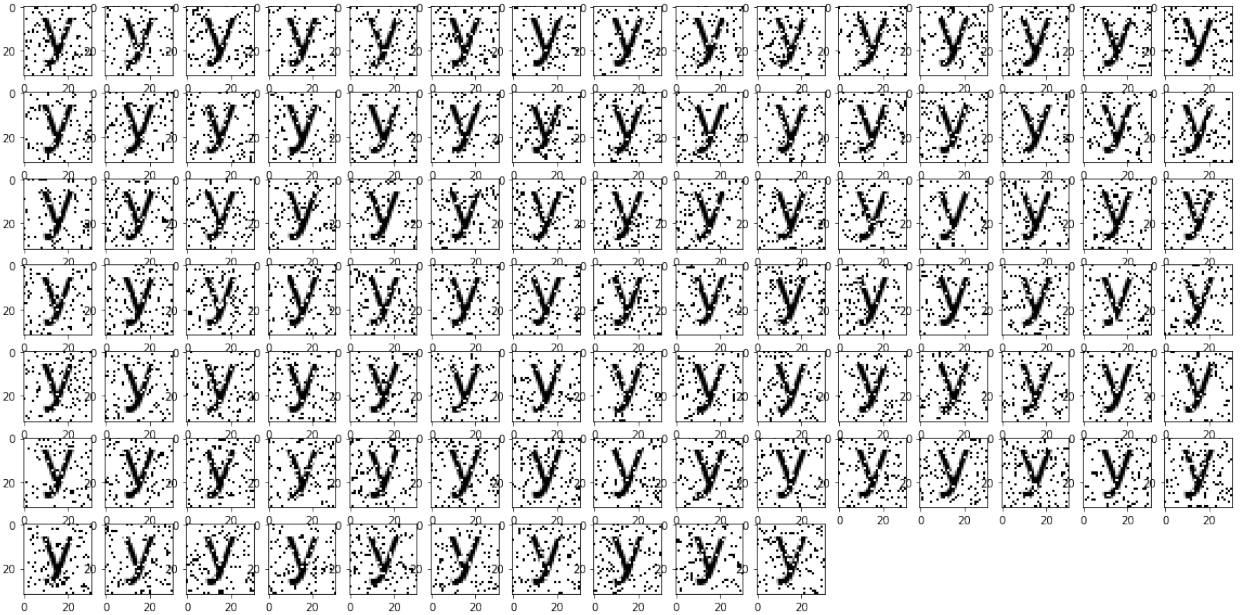
```
Training for label w...
```



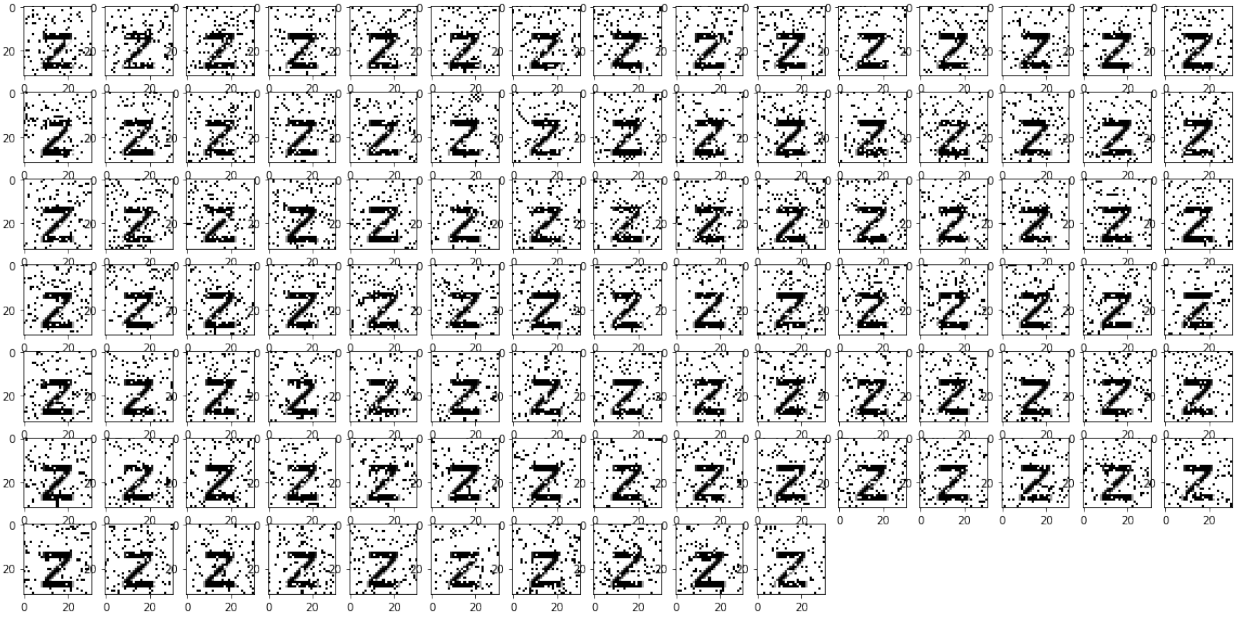
```
Training for label x...
```



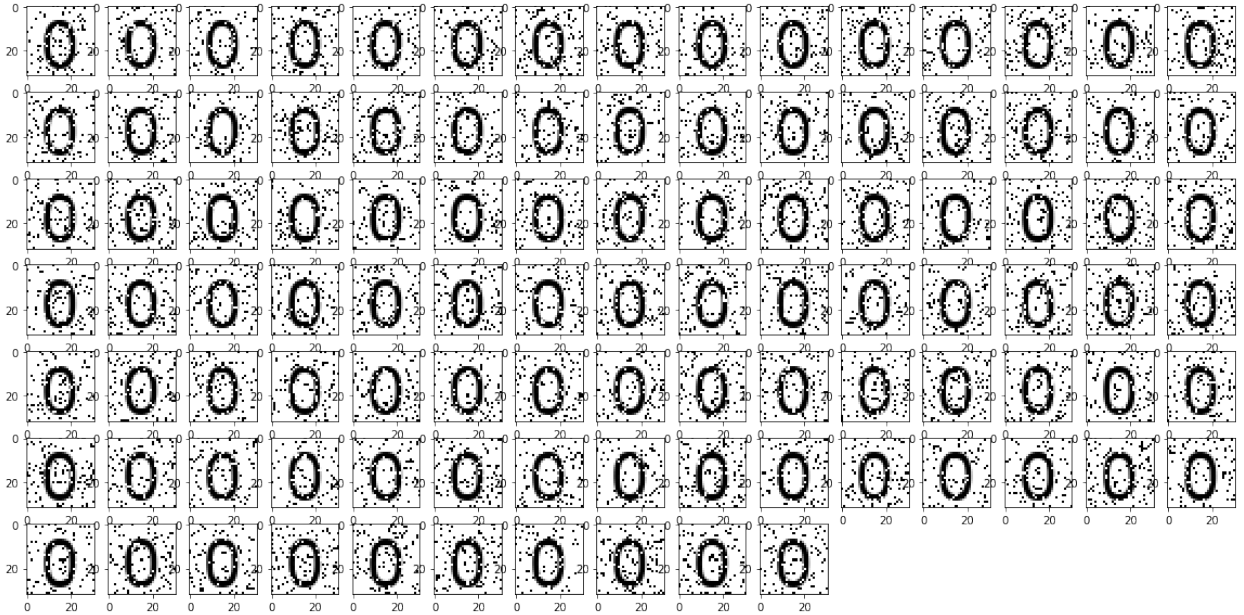
```
Training for label y...
```



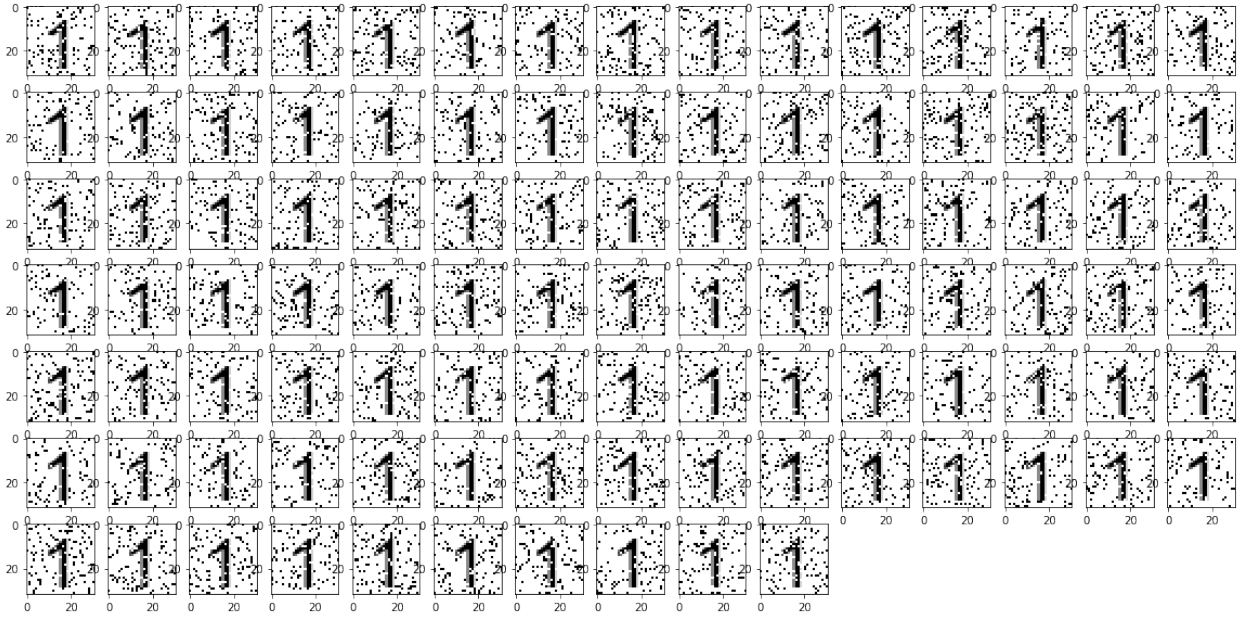
```
Training for label z...
```



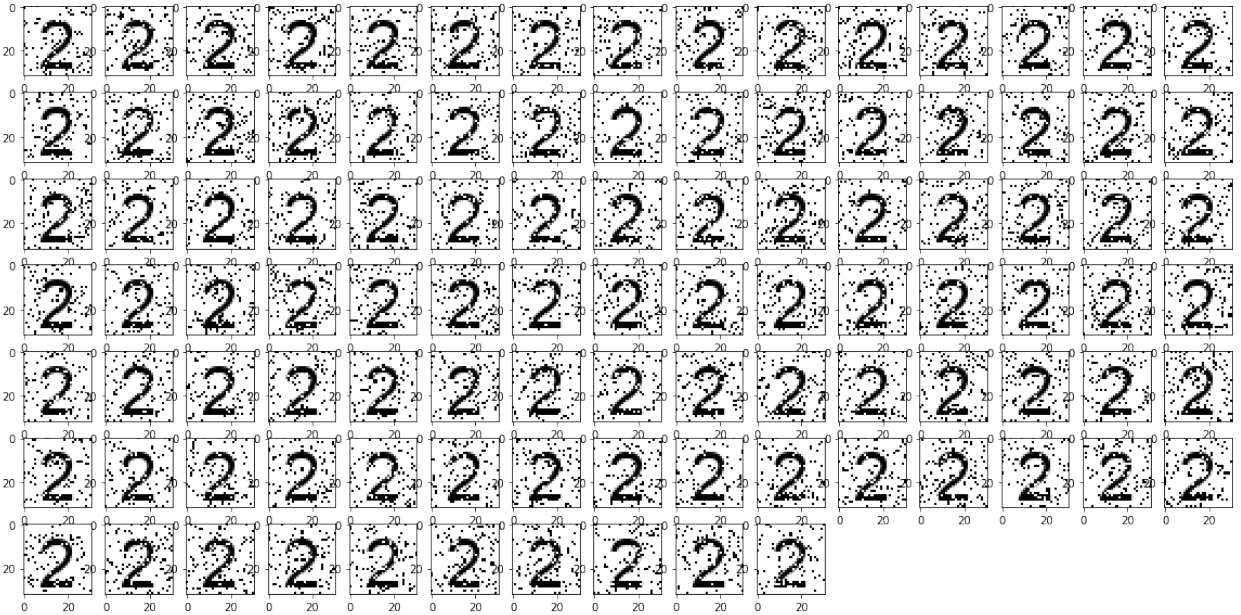
```
Training for label 0...
```



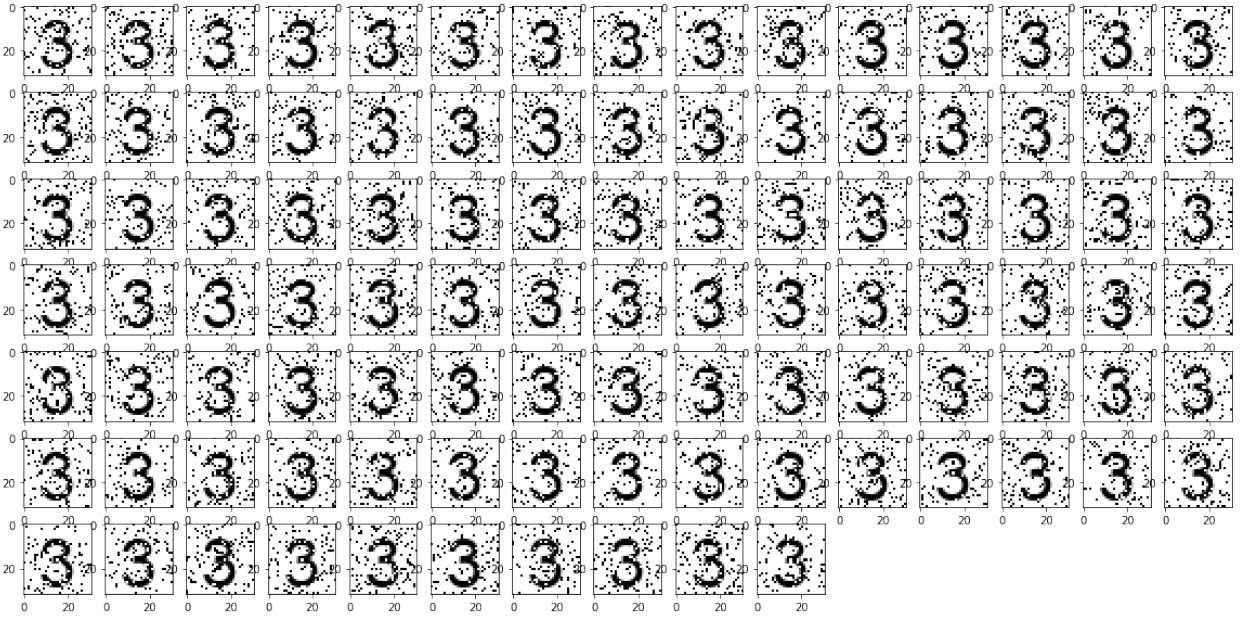
```
Training for label 1...
```



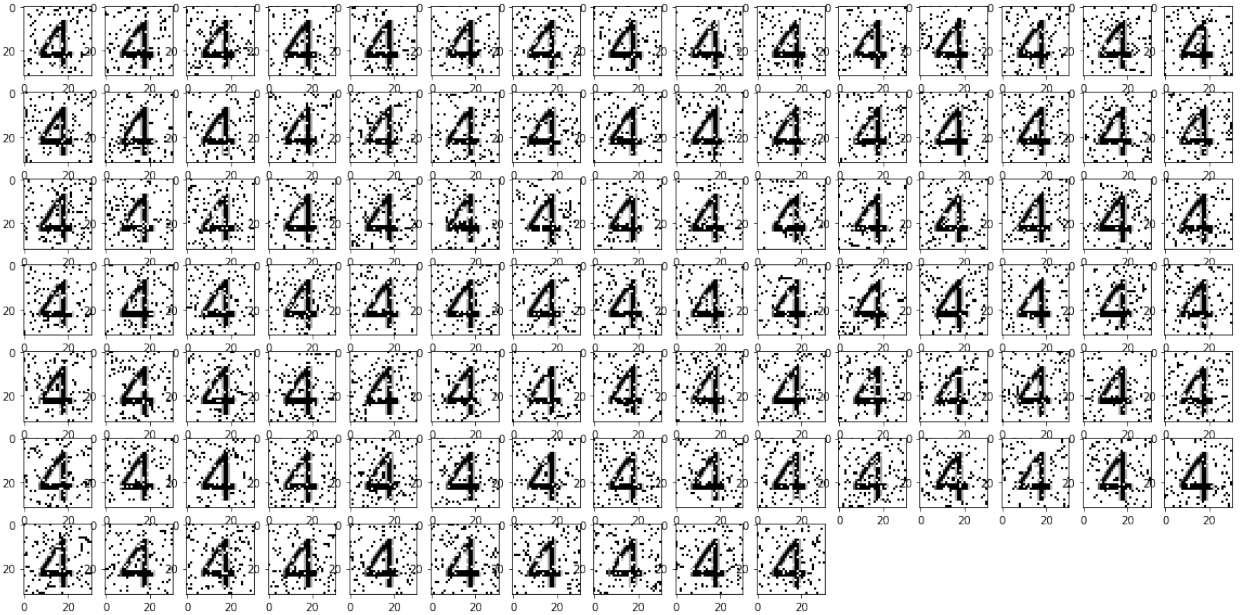
```
Training for label 2...
```



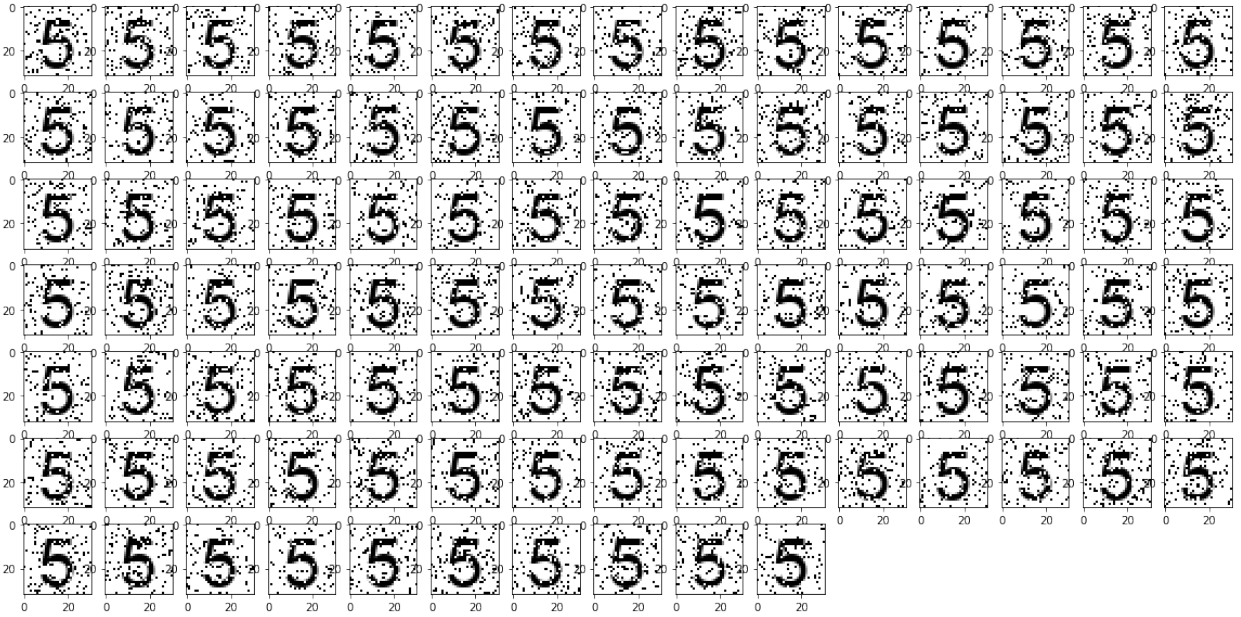
```
Training for label 3...
```



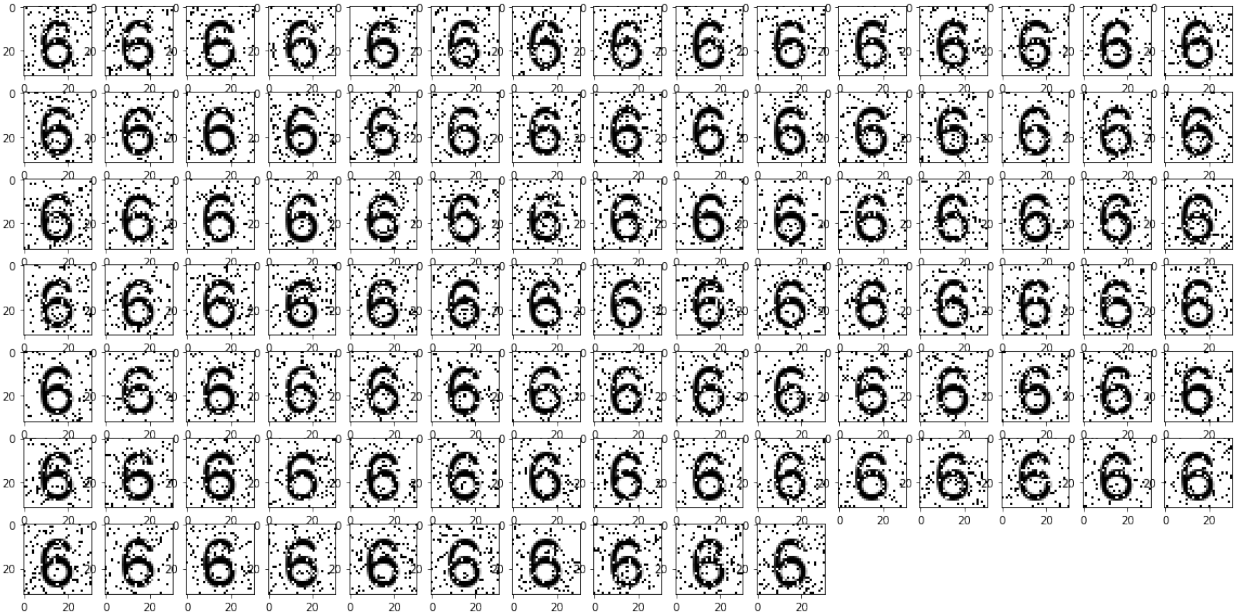
```
Training for label 4...
```



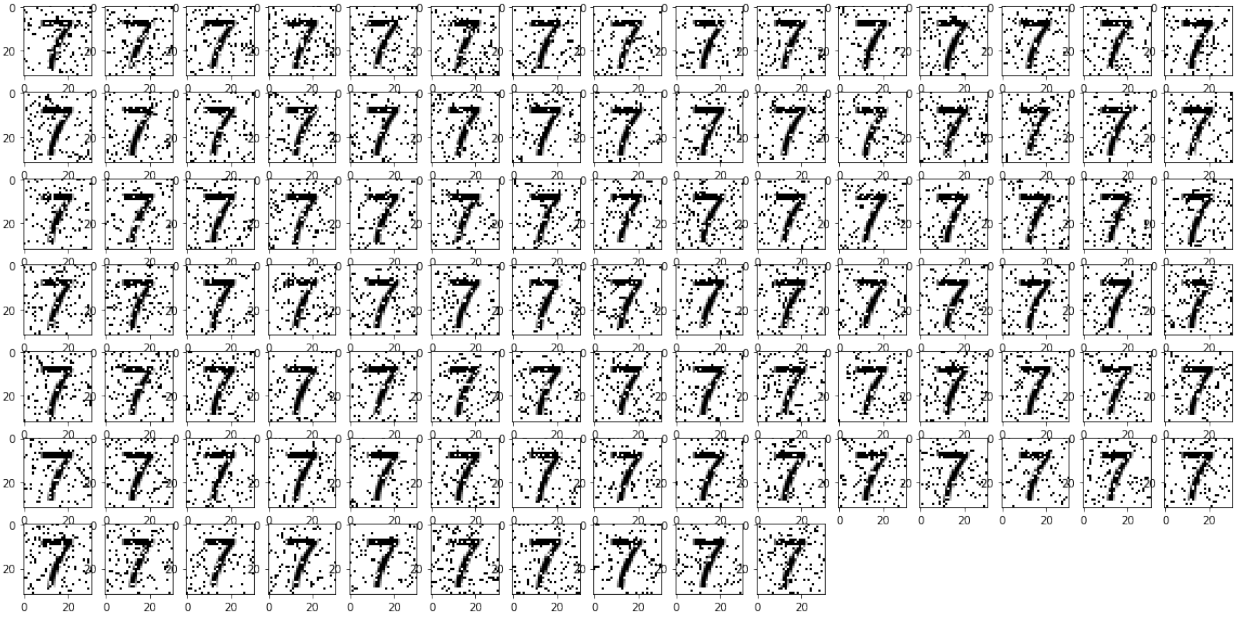
```
Training for label 5...
```



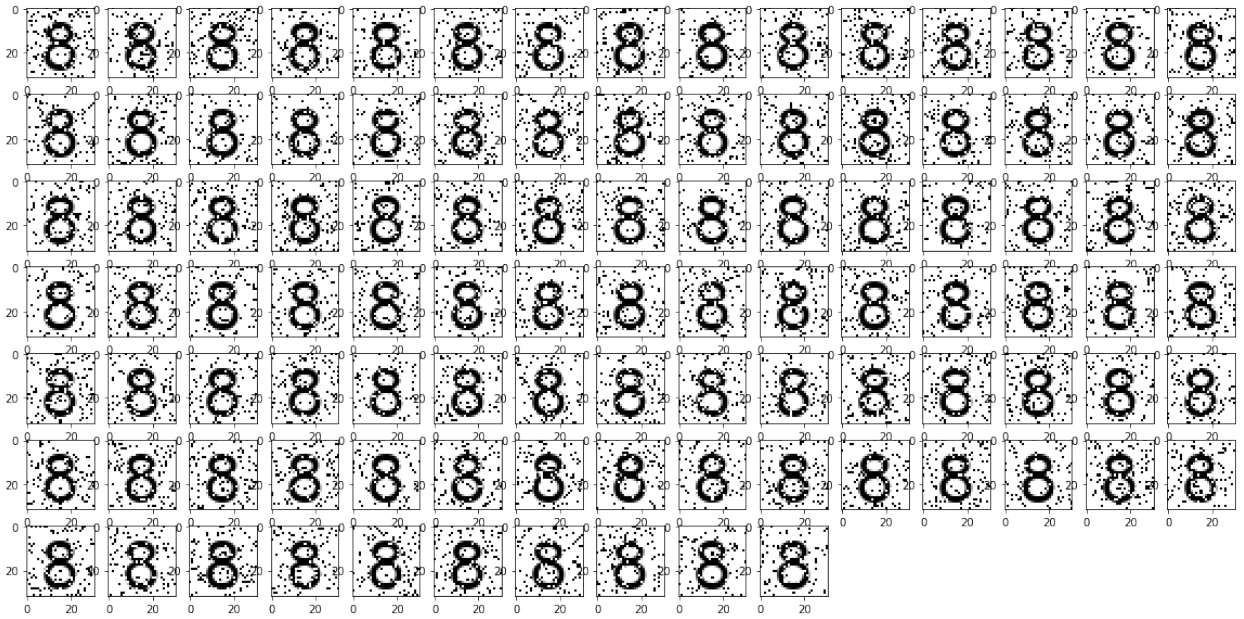
```
Training for label 6...
```



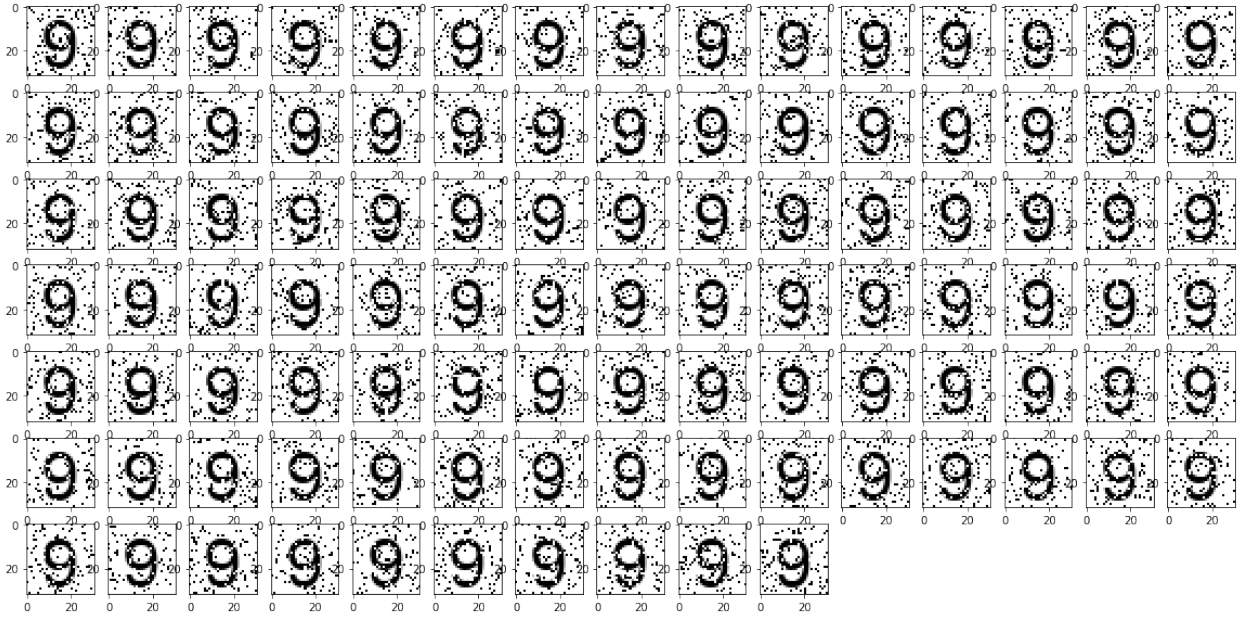
```
Training for label 7...
```



```
Training for label 8...
```

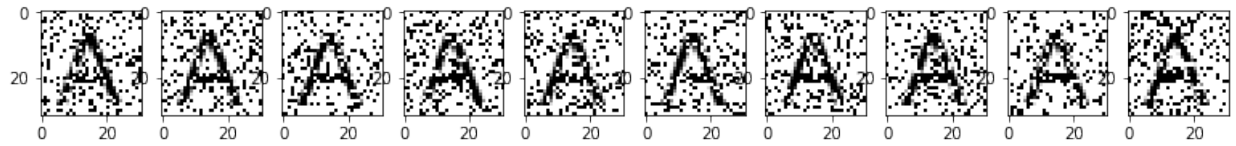


```
Training for label 9...
```

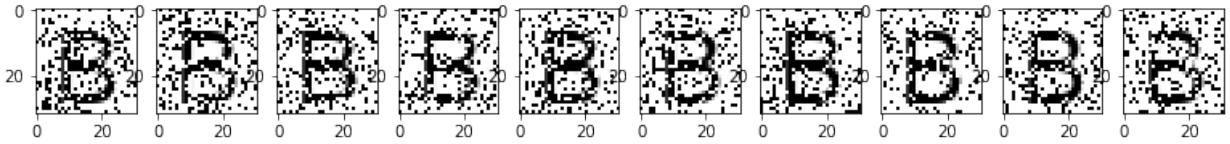


Testing with high noise

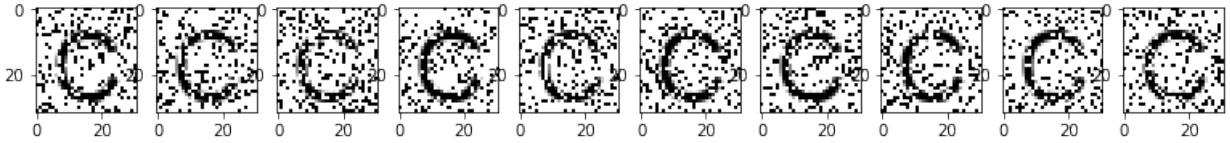
```
In [16]: for x in 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789':
             iter_read(x)
```



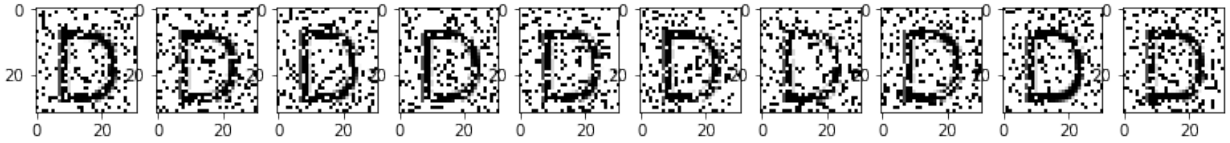
```
!! A correct=10 wrong= 0 answers=['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']
```



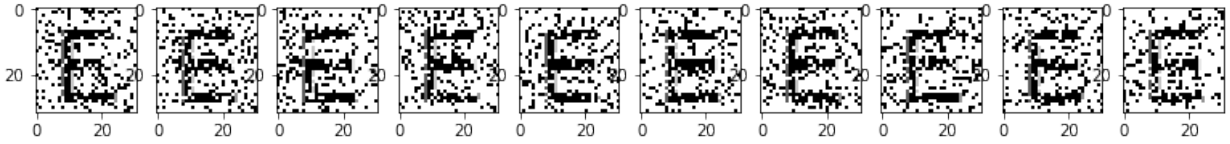
```
!! B correct= 9 wrong= 1 answers=['S', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B']
```



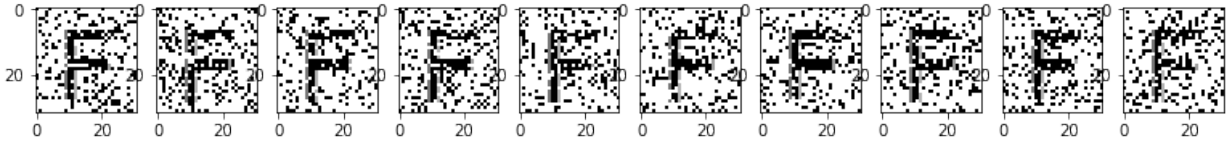
```
!! C correct=10 wrong= 0 answers=['C', 'C', 'C', 'C', 'C', 'C', 'C', 'C', 'C', 'C']
```



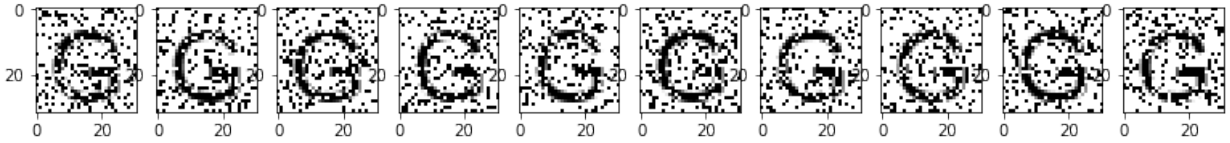
```
!! D correct=10 wrong= 0 answers=['D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D']
```



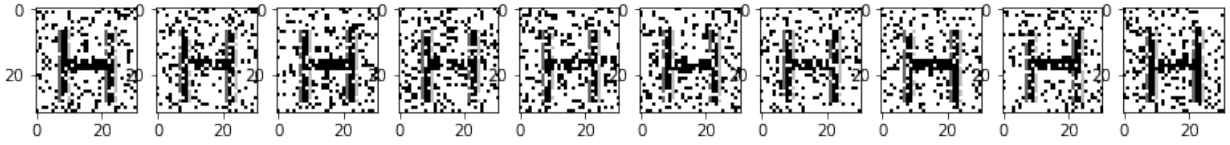
```
!! E correct=10 wrong= 0 answers=['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E']
```



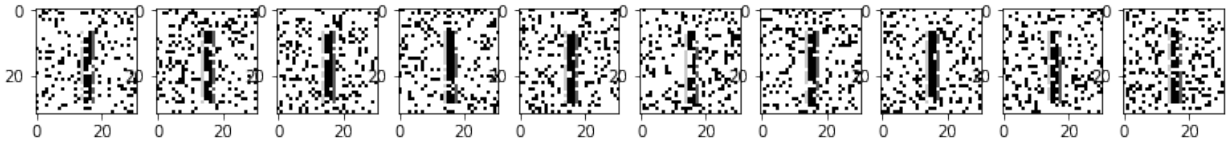
```
!! F correct=10 wrong= 0 answers=['F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F']
```



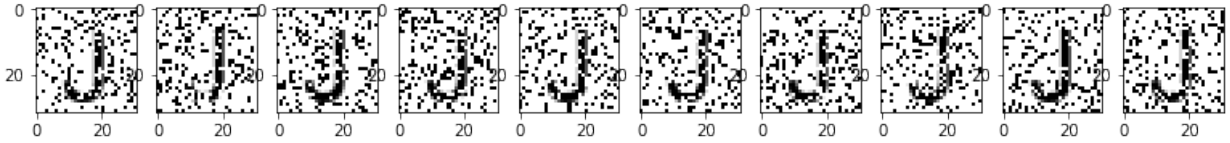
```
!! G correct=10 wrong= 0 answers=['G', 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'G']
```



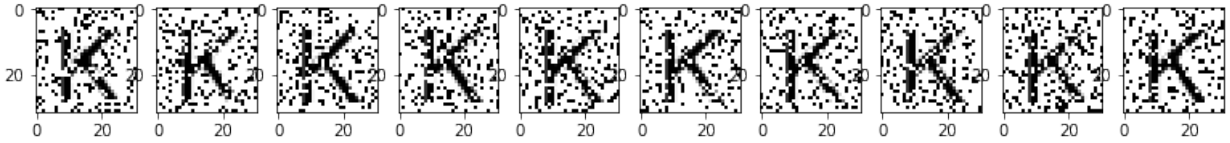
```
!! H correct=10 wrong= 0 answers=['H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H']
```



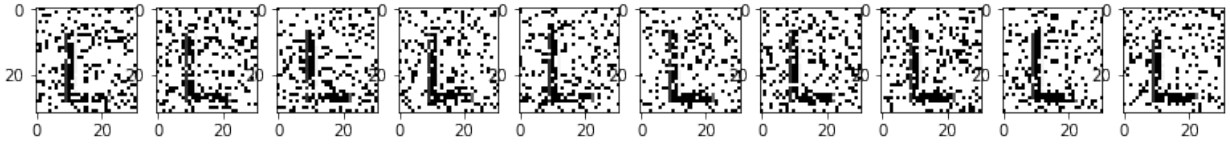
```
!! I correct=10 wrong= 0 answers=['I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I']
```



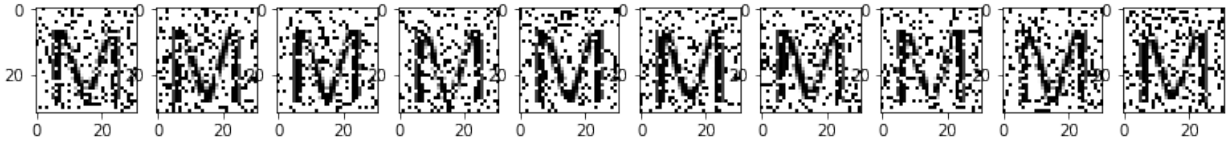
```
!! J correct=10 wrong= 0 answers=['J', 'J', 'J', 'J', 'J', 'J', 'J', 'J', 'J', 'J']
```



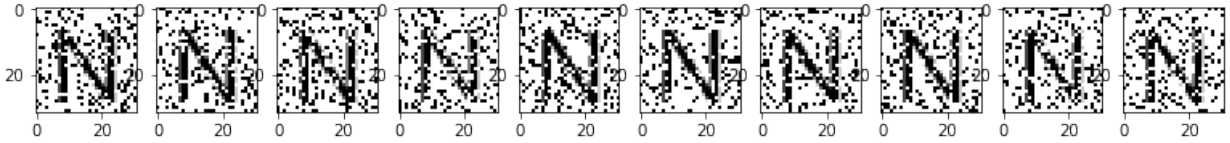
```
!! K correct=10 wrong= 0 answers=['K', 'K', 'K', 'K', 'K', 'K', 'K', 'K', 'K', 'K']
```



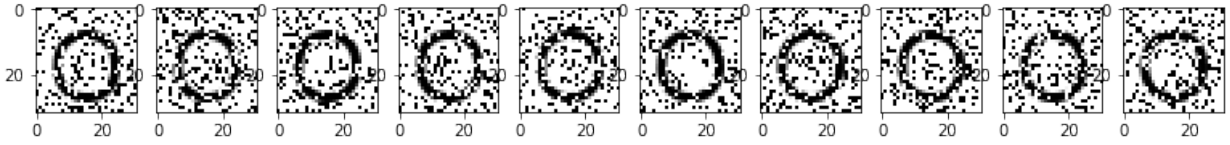
```
!! L correct=10 wrong= 0 answers=['L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
```



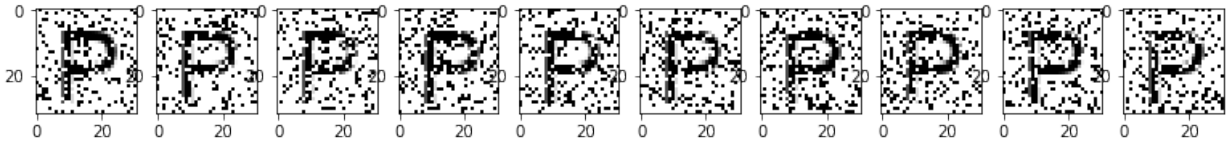
```
!! M correct=10 wrong= 0 answers=['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']
```



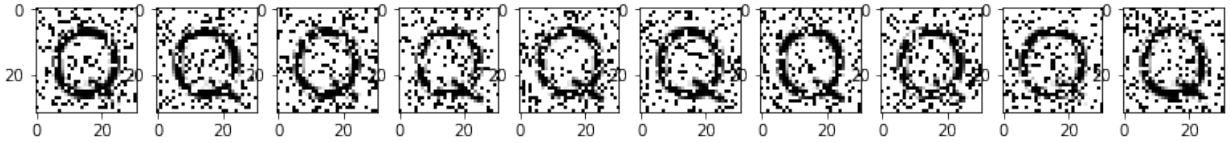
```
!! N correct=10 wrong= 0 answers=['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']
```



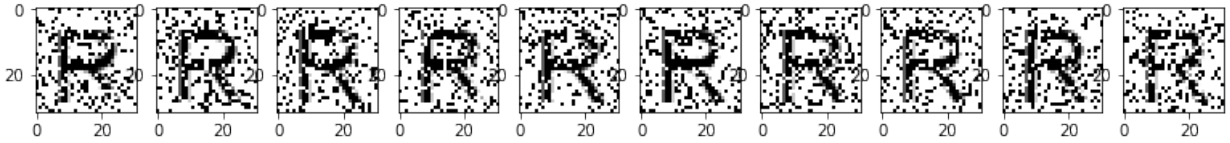
```
!! O correct= 8 wrong= 2 answers=['G', 'G', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```



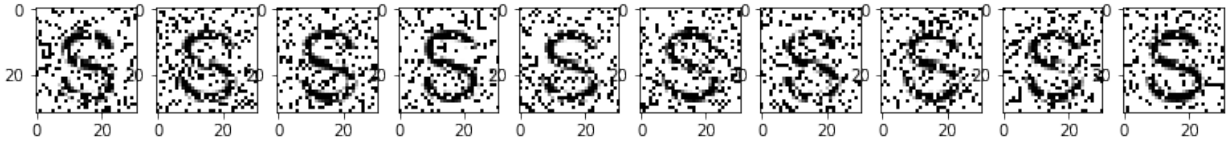
```
!! P correct=10 wrong= 0 answers=['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
```



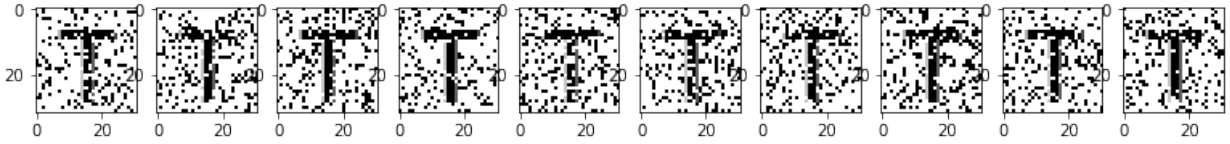
```
!! Q correct=10 wrong= 0 answers=['Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q']
```



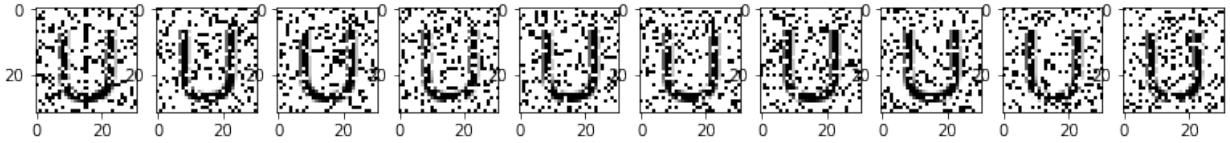
```
!! R correct=10 wrong= 0 answers=['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
```



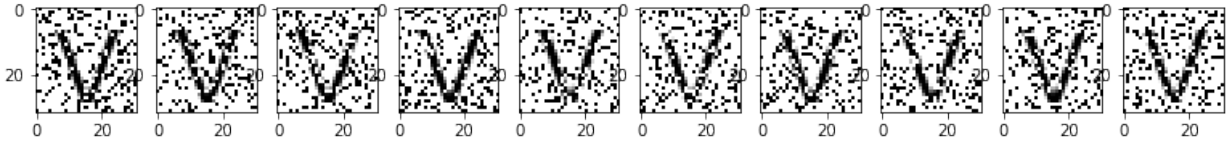
```
!! S correct=10 wrong= 0 answers=['S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S']
```



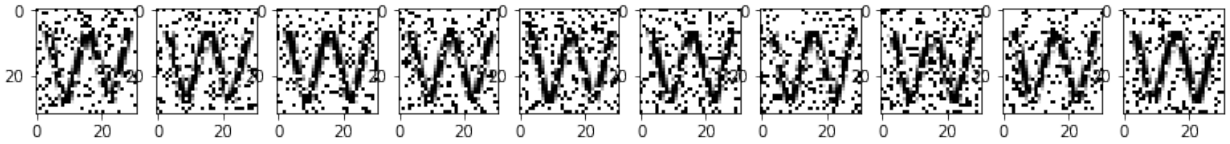
```
!! T correct= 9 wrong= 1 answers=['T', 'T', 'T', 'T', 'T', 'I', 'T', 'T', 'T', 'T']
```



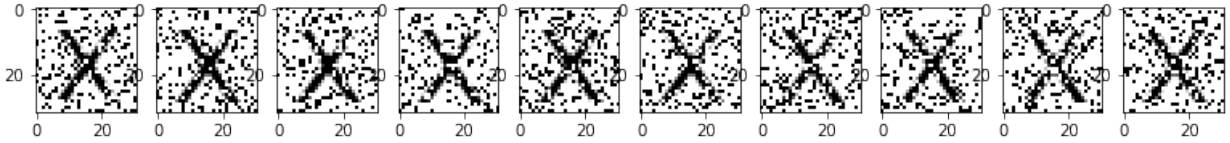
```
!! U correct=10 wrong= 0 answers=['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U']
```



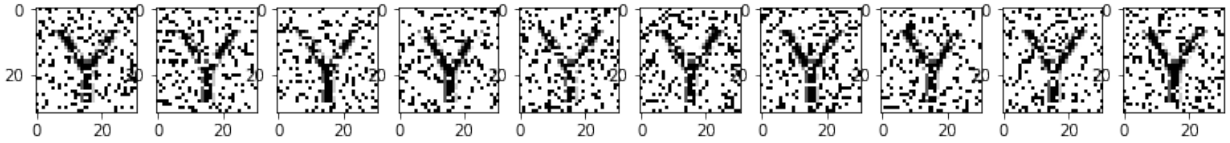
```
!! V correct=10 wrong= 0 answers=['V', 'V', 'V', 'V', 'V', 'V', 'V', 'V', 'V', 'V']
```



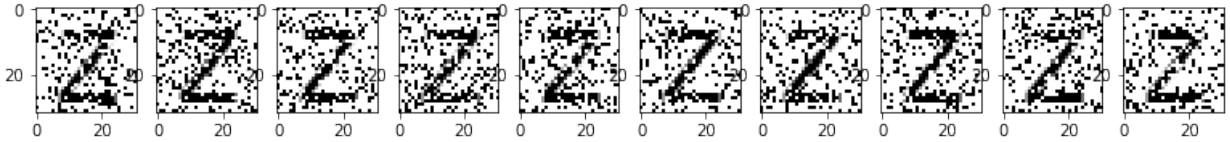
```
!! W correct=10 wrong= 0 answers=['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
```



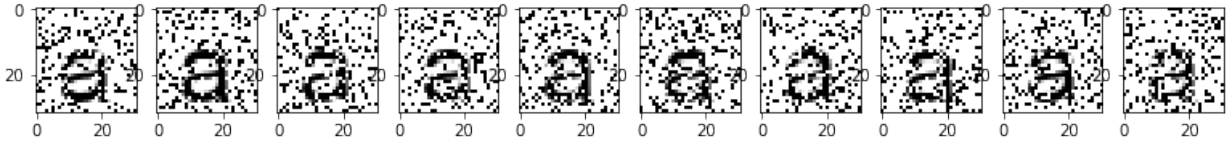
```
!! X correct=10 wrong= 0 answers=['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
```



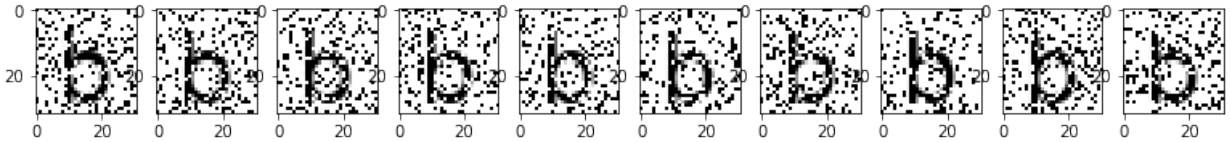
```
!! Y correct= 9 wrong= 1 answers=['Y', 'I', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y']
```



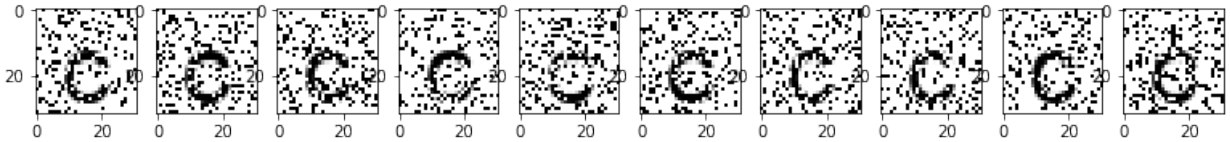
```
!! Z correct=10 wrong= 0 answers=['Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z']
```



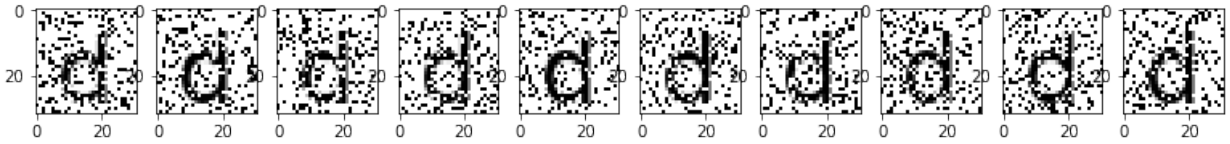
```
!! a correct=10 wrong= 0 answers=['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```



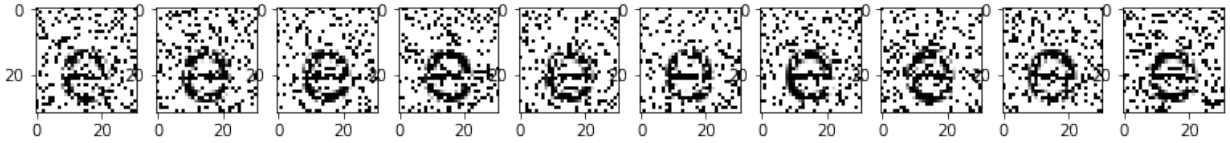
```
!! b correct= 3 wrong= 7 answers=['o', 'o', 'o', 'b', 'o', 'h', 'h', 'b', 'b', 'o']
```



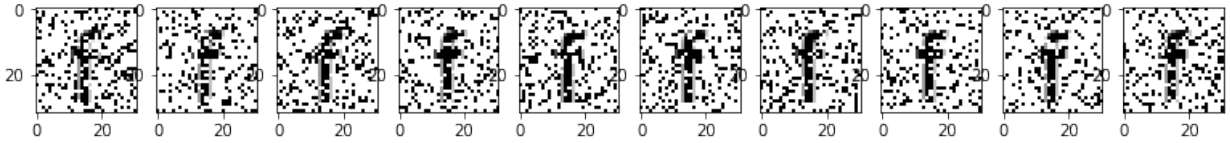
```
!! c correct= 8 wrong= 2 answers=['c', 'c', 'c', 'c', 'c', 'o', 'c', 'c', 'c', 'o']
```



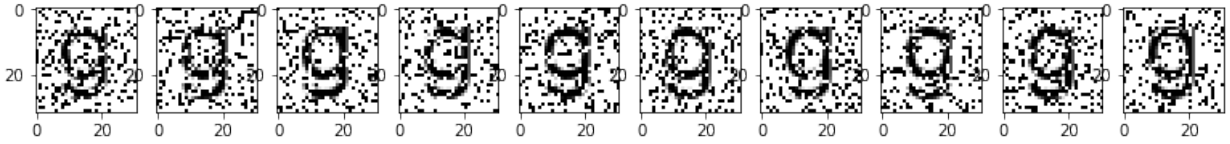
```
!! d correct=10 wrong= 0 answers=['d', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd']
```



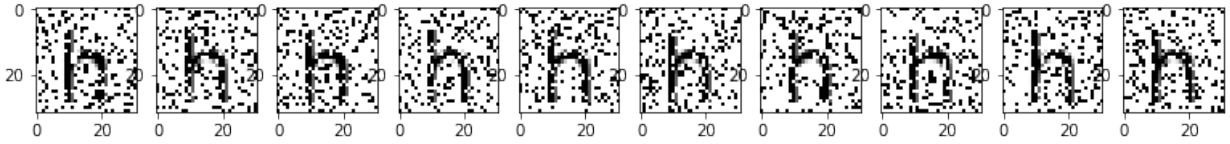
```
!! e correct= 4 wrong= 6 answers=['e', 'o', 'e', 'o', 'o', 'o', 'e', 'o', 'o', 'e']
```



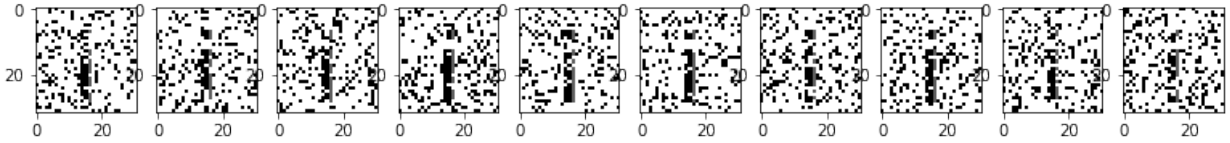
```
!! f correct= 0 wrong=10 answers=['I', 'I', 'I', 'I', 'i', 'I', 'I', 'I', 'I', 'I']
```



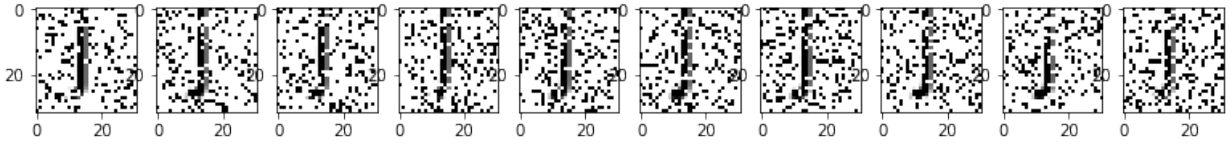
```
!! g correct=10 wrong= 0 answers=['g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g']
```



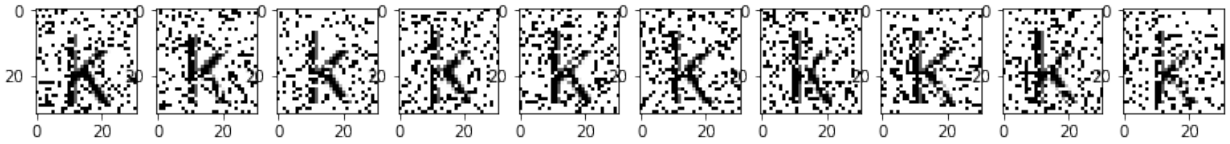
```
!! h correct=10 wrong= 0 answers=['h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h']
```



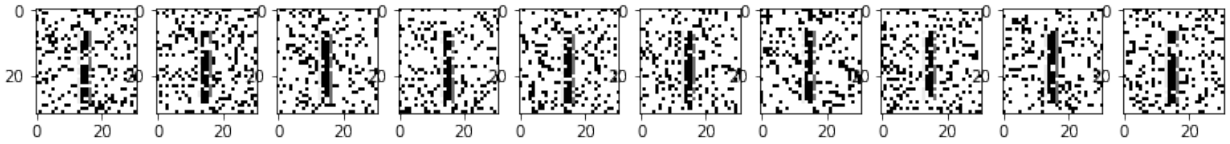
```
!! i correct= 8 wrong= 2 answers=['i', 'i', 'i', 'I', 'i', 'i', 'i', 'i', 'I', 'i']
```



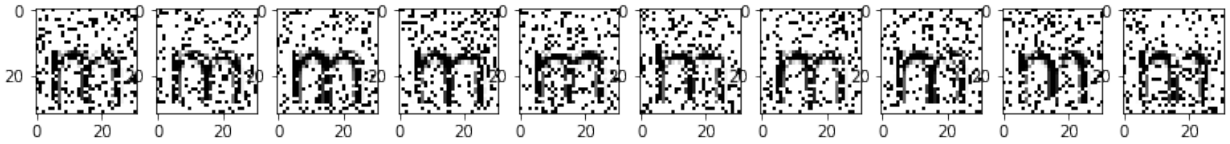
```
!! j correct= 7 wrong= 3 answers=['j', 'j', 'j', 'I', 'I', 'j', 'j', 'j', 'j', 'I']
```



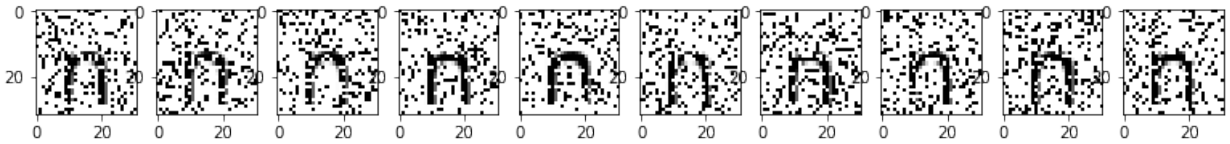
```
!! k correct=10 wrong= 0 answers=['k', 'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k']
```



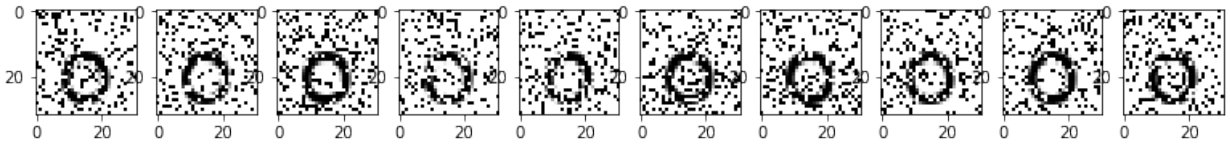
```
!! l correct= 0 wrong=10 answers=['I', 'i', 'I', 'I', 'I', 'I', 'i', 'I', 'I', 'i']
```



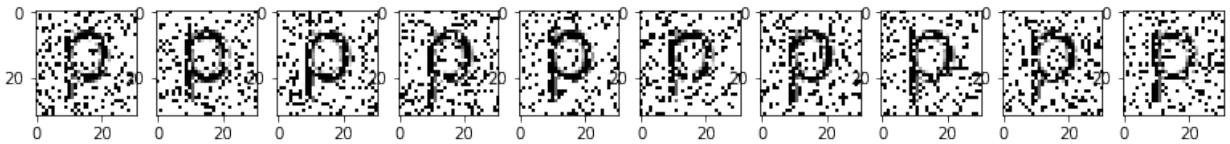
```
!! m correct=10 wrong= 0 answers=['m', 'm', 'm', 'm', 'm', 'm', 'm', 'm', 'm', 'm']
```



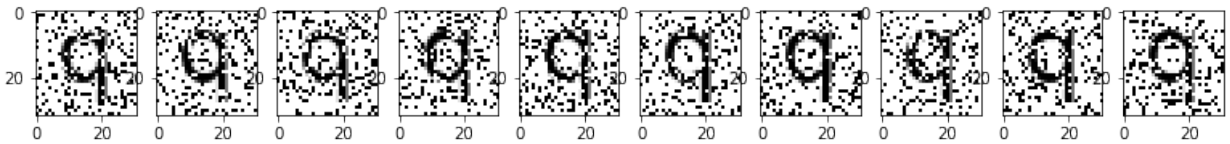
```
!! n correct= 5 wrong= 5 answers=['u', 'n', 'n', 'n', 'n', 'n', 'u', 'u', 'u', 'h']
```



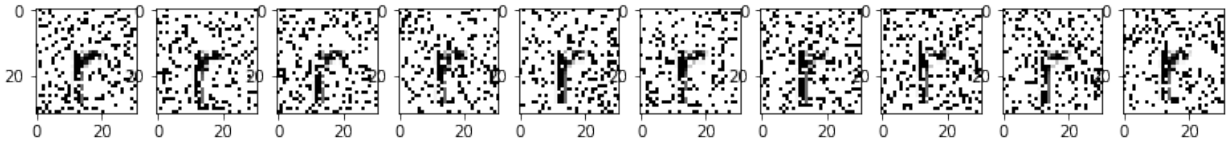
```
!! o correct=10 wrong= 0 answers=['o', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'o']
```



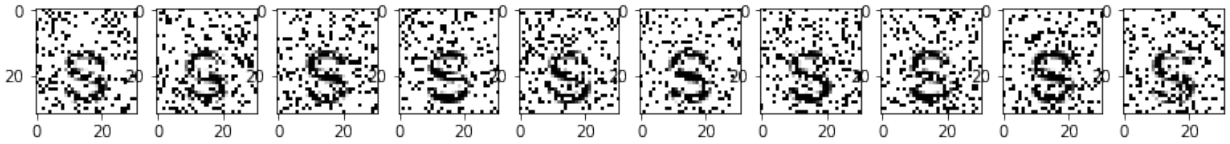
```
!! p correct=10 wrong= 0 answers=['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p']
```



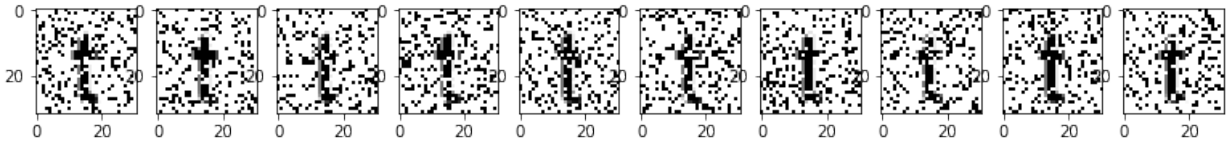
```
!! q correct= 9 wrong= 1 answers=['q', 'q', 'q', 'q', 'q', 'q', 'q', 'q', 'q', 'g']
```



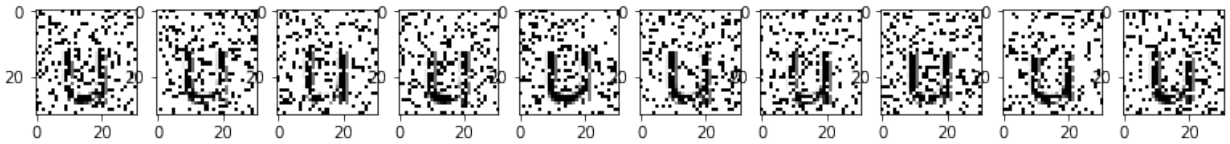
```
!! r correct=10 wrong= 0 answers=['r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r']
```



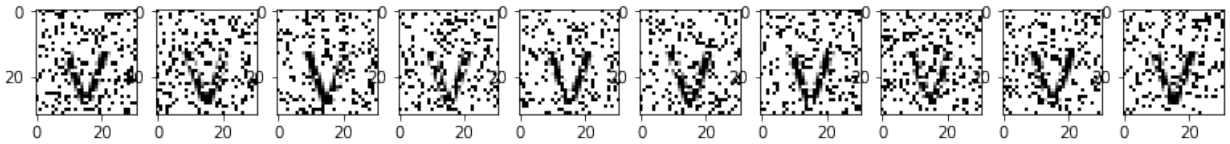
```
!! s correct=10 wrong= 0 answers=['s', 's', 's', 's', 's', 's', 's', 's', 's', 's']
```



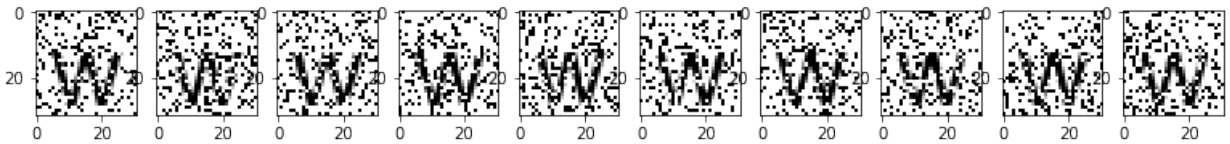
```
!! t correct= 0 wrong=10 answers=['I', 'r', 'I', 'i', 'I', 'i', 'i', 'i', 'I', 'i']
```



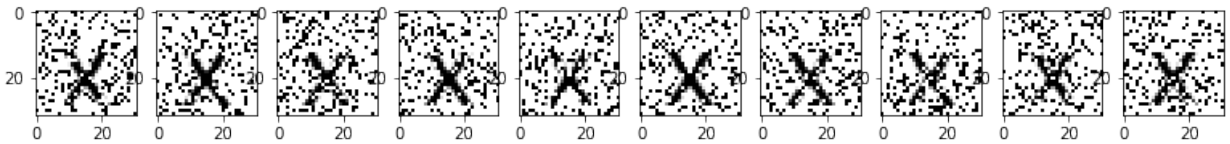
```
!! u correct=10 wrong= 0 answers=['u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u']
```



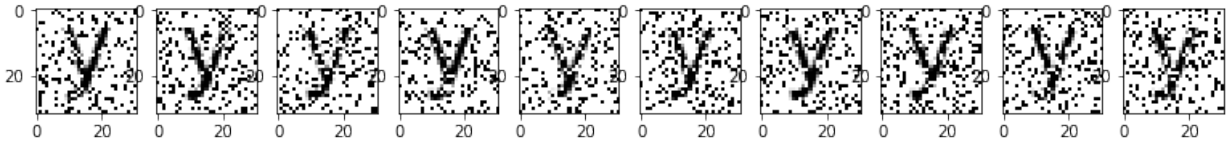
```
!! v correct=10 wrong= 0 answers=['v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v']
```



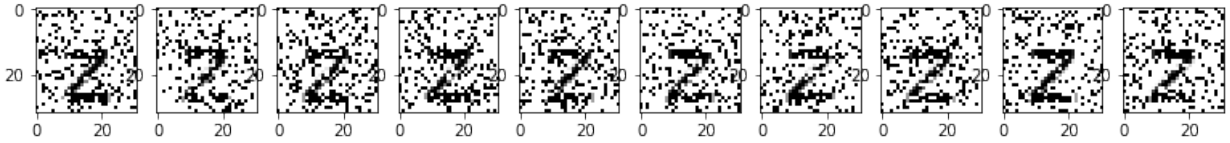
```
!! w correct=10 wrong= 0 answers=['w', 'w', 'w', 'w', 'w', 'w', 'w', 'w', 'w', 'w']
```



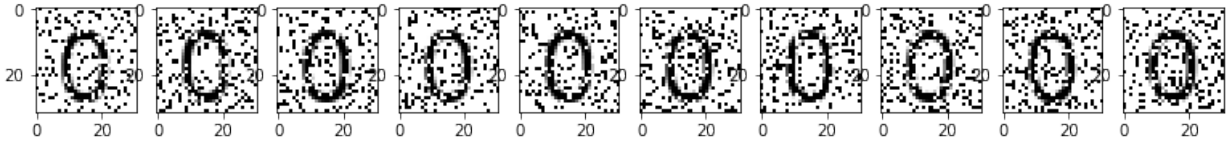
```
!! x correct=10 wrong= 0 answers=['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']
```



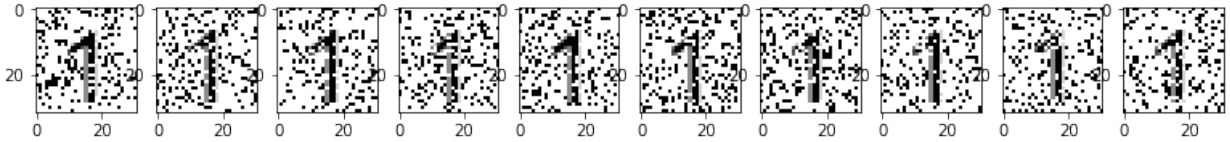
```
!! y correct=10 wrong= 0 answers=['y', 'y', 'y', 'y', 'y', 'y', 'y', 'y', 'y', 'y']
```



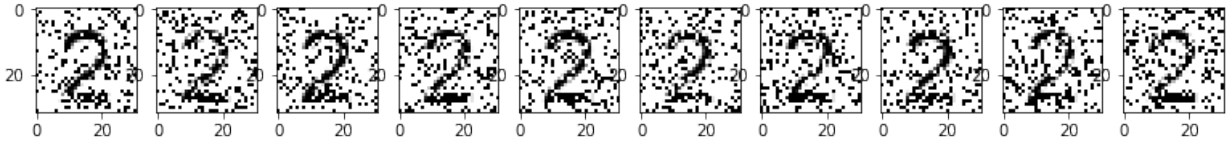
```
!! z correct=10 wrong= 0 answers=['z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z']
```



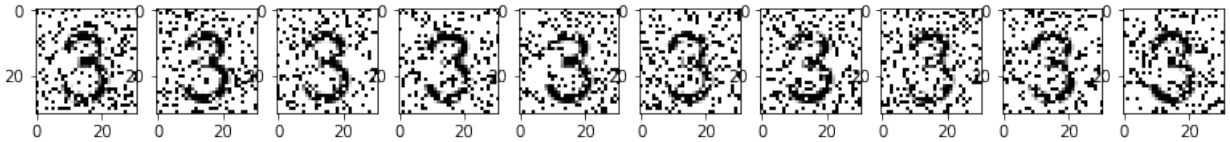
```
!! 0 correct=10 wrong= 0 answers=['0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
```



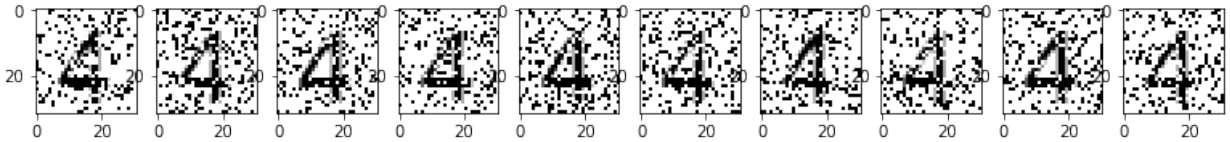
```
!! 1 correct= 5 wrong= 5 answers=['1', 'I', '1', 'I', '1', '1', 'I', 'I', '1', 'I']
```



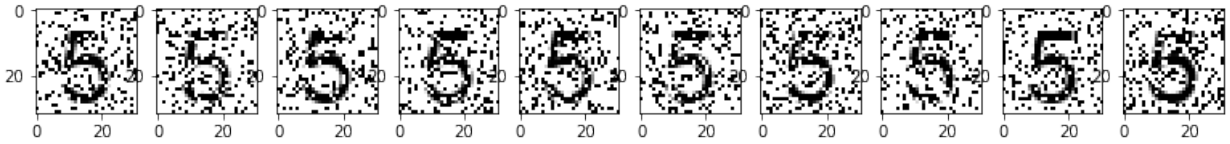
```
!! 2 correct=10 wrong= 0 answers=['2', '2', '2', '2', '2', '2', '2', '2', '2', '2']
```



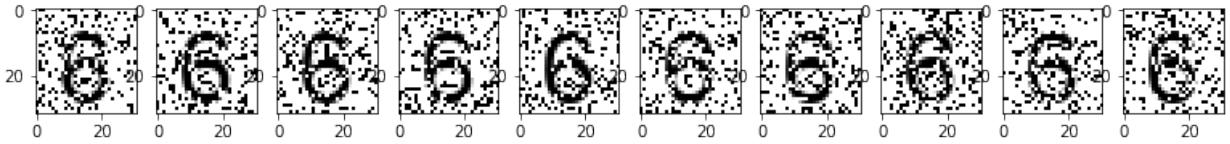
```
!! 3 correct=10 wrong= 0 answers=['3', '3', '3', '3', '3', '3', '3', '3', '3', '3']
```



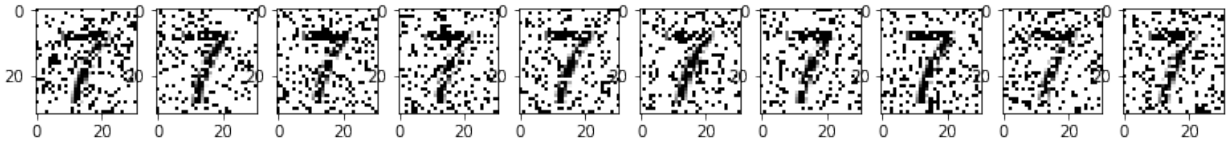
```
!! 4 correct=10 wrong= 0 answers=['4', '4', '4', '4', '4', '4', '4', '4', '4', '4']
```



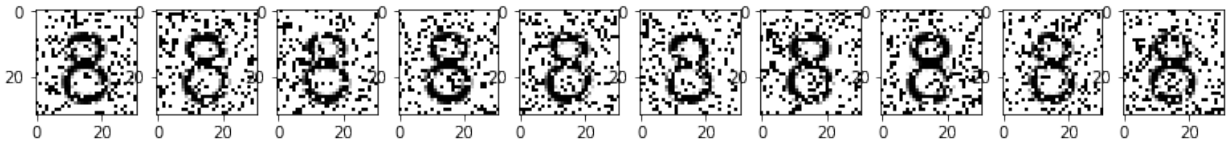
```
!! 5 correct=10 wrong= 0 answers=['5', '5', '5', '5', '5', '5', '5', '5', '5', '5']
```



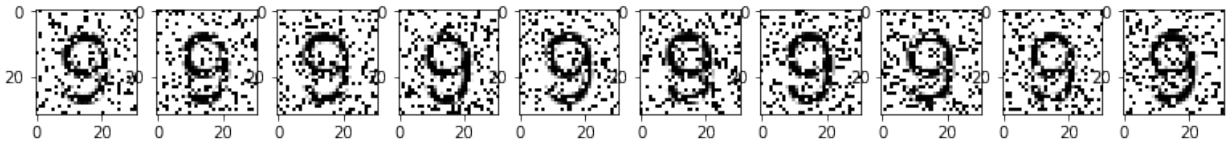
```
!! 6 correct=10 wrong= 0 answers=['6', '6', '6', '6', '6', '6', '6', '6', '6', '6']
```



```
!! 7 correct= 7 wrong= 3 answers=['7', '7', '7', 'I', '7', 'I', 'I', '7', '7', '7']
```



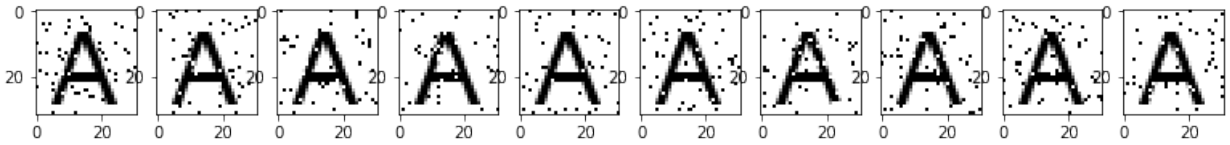
```
!! 8 correct= 4 wrong= 6 answers=['8', '6', '6', '6', '8', 'd', '8', '8', 'd', '6']
```



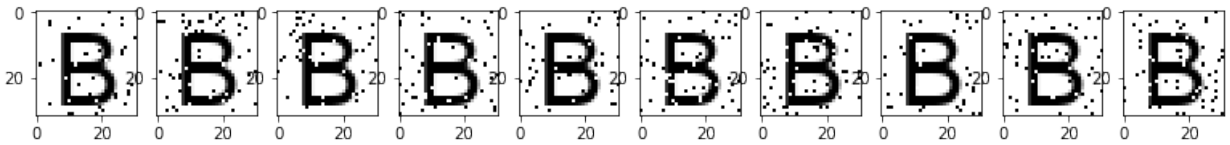
```
!! 9 correct= 3 wrong= 7 answers=['9', '0', '6', '0', '9', '0', '0', '9', '0', '0']
```

Testing with low noise

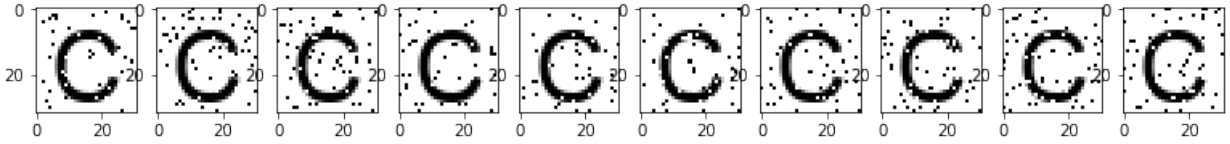
```
In [17]: for x in 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789':
             iter_read(x, p=0.05)
```



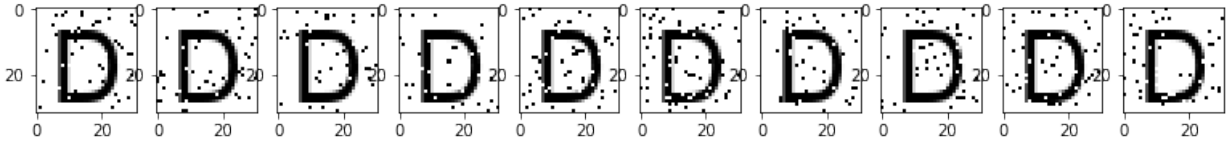
```
!! A correct=10 wrong= 0 answers=['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']
```



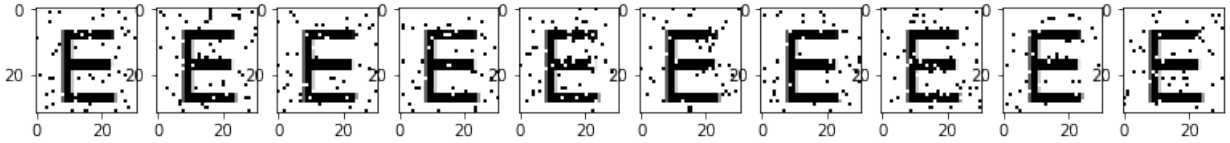
```
!! B correct=10 wrong= 0 answers=['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B']
```



```
!! C correct=10 wrong= 0 answers=['C', 'C', 'C', 'C', 'C', 'C', 'C', 'C', 'C', 'C']
```

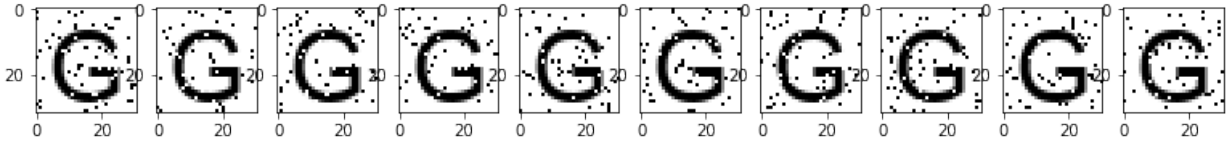


```
!! D correct=10 wrong= 0 answers=['D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D']
```

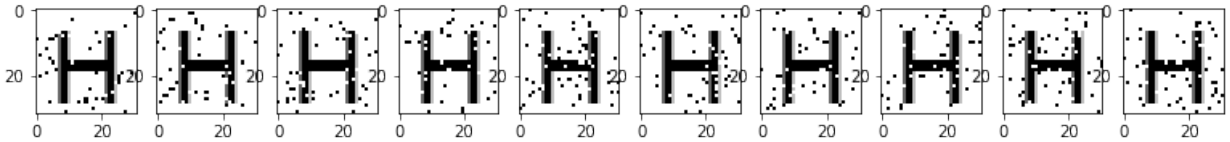


```
!! E correct=10 wrong= 0 answers=['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E']
```

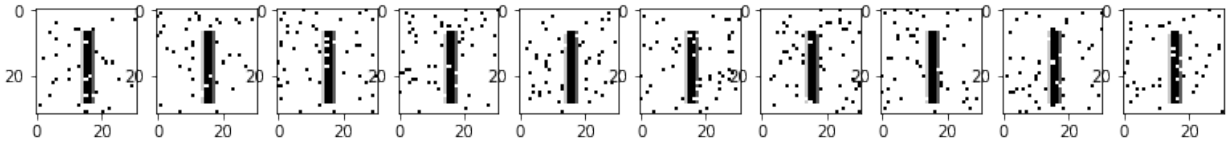
```
!! F correct=10 wrong= 0 answers=['F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F']
```



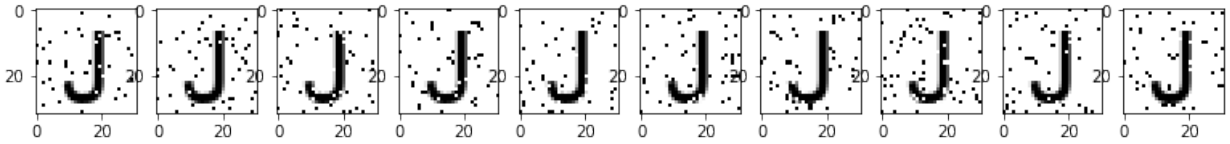
```
!! G correct=10 wrong= 0 answers=['G', 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'G']
```



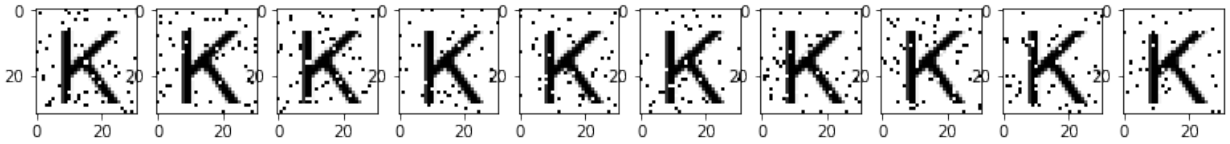
```
!! H correct=10 wrong= 0 answers=['H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H']
```



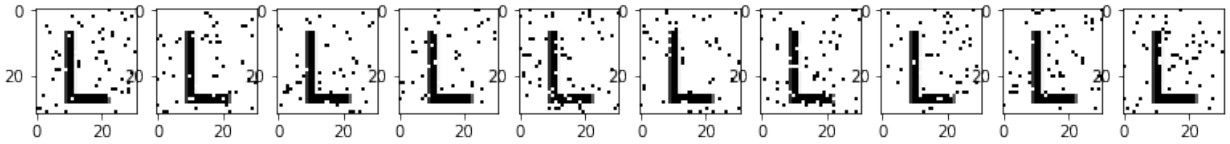
```
!! I correct=10 wrong= 0 answers=['I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I']
```



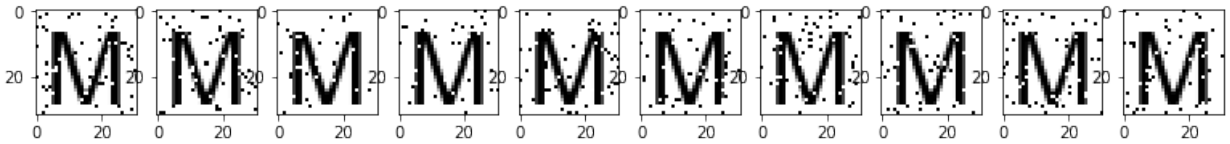
```
!! J correct=10 wrong= 0 answers=['J', 'J', 'J', 'J', 'J', 'J', 'J', 'J', 'J', 'J']
```



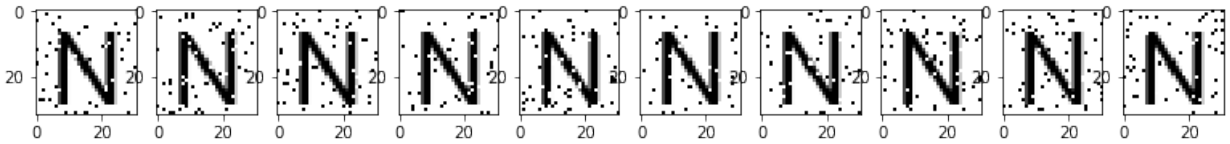
```
!! K correct=10 wrong= 0 answers=['K', 'K', 'K', 'K', 'K', 'K', 'K', 'K', 'K', 'K']
```



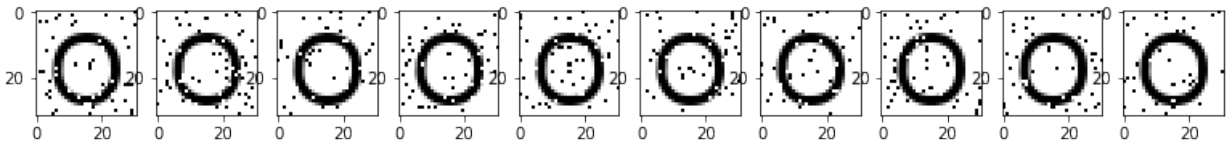
```
!! L correct=10 wrong= 0 answers=['L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
```



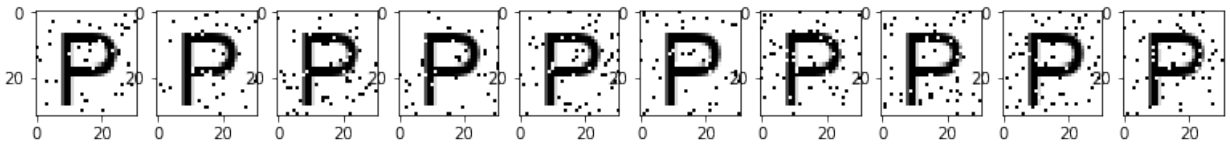
```
!! M correct=10 wrong= 0 answers=['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']
```



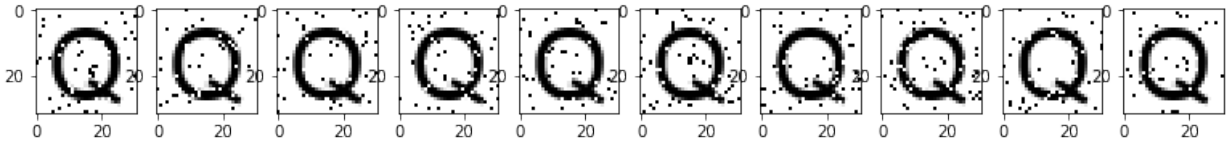
```
!! N correct=10 wrong= 0 answers=['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']
```



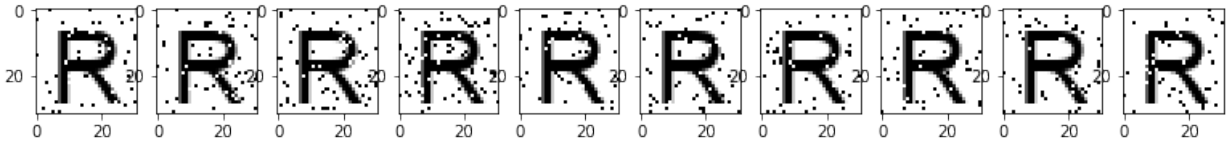
```
!! O correct=10 wrong= 0 answers=['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```



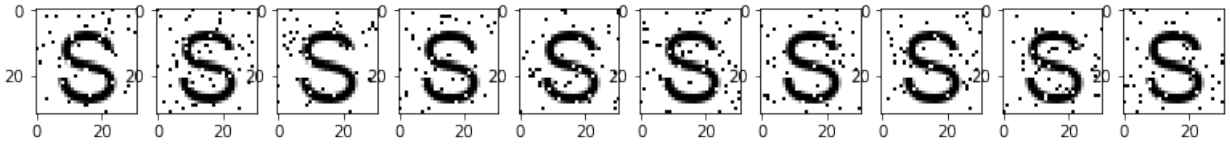
```
!! P correct=10 wrong= 0 answers=['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P']
```



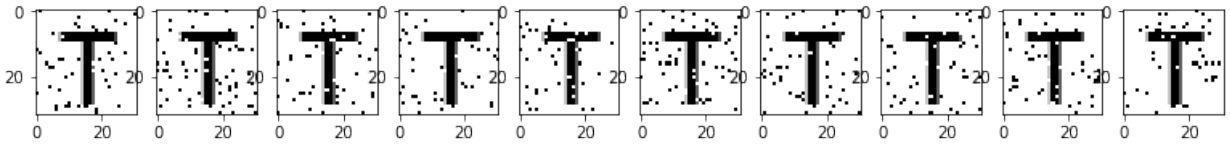
```
!! Q correct=10 wrong= 0 answers=['Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q']
```



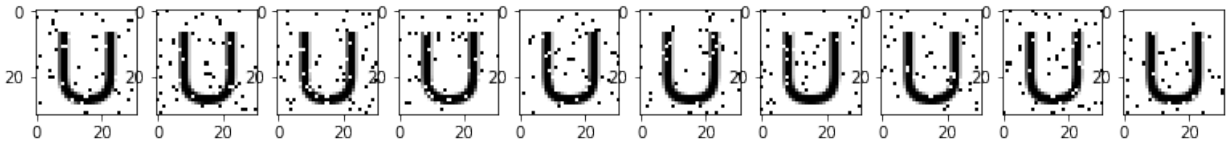
```
!! R correct=10 wrong= 0 answers=['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
```



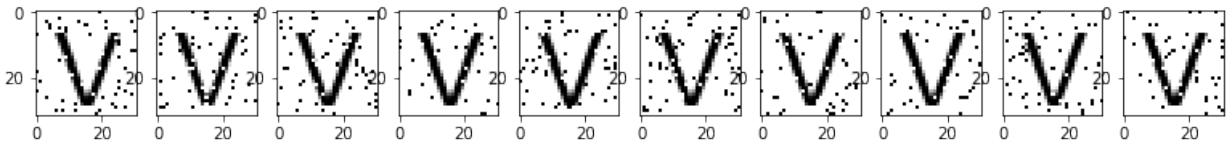
```
!! S correct=10 wrong= 0 answers=['S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S']
```



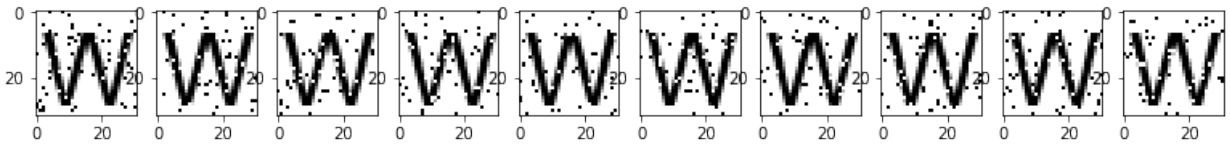
```
!! T correct=10 wrong= 0 answers=['T', 'T', 'T', 'T', 'T', 'T', 'T', 'T', 'T', 'T']
```



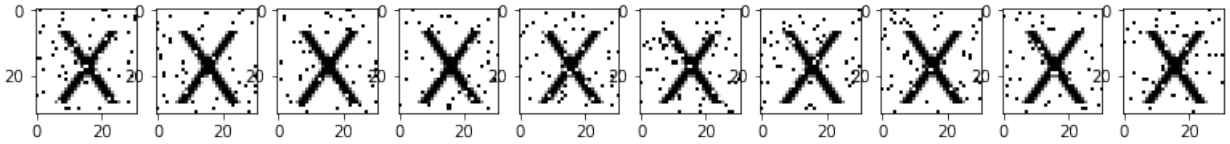
```
!! U correct=10 wrong= 0 answers=['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U']
```



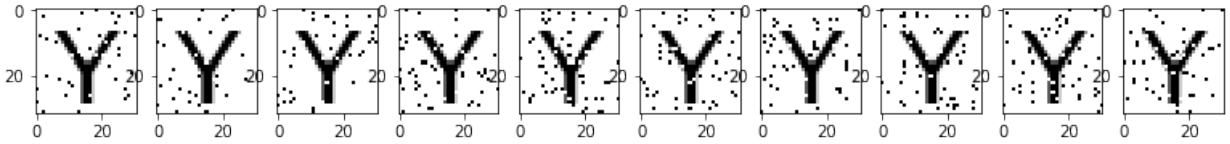
```
!! V correct=10 wrong= 0 answers=['V', 'V', 'V', 'V', 'V', 'V', 'V', 'V', 'V', 'V']
```



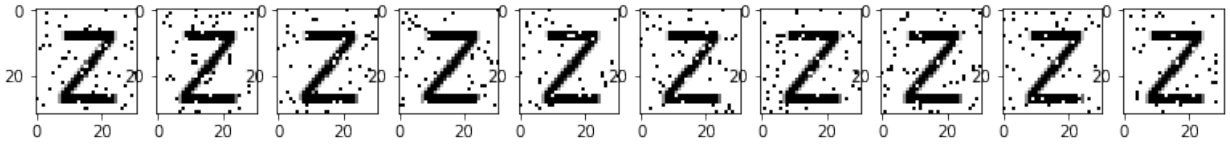
```
!! W correct=10 wrong= 0 answers=['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
```



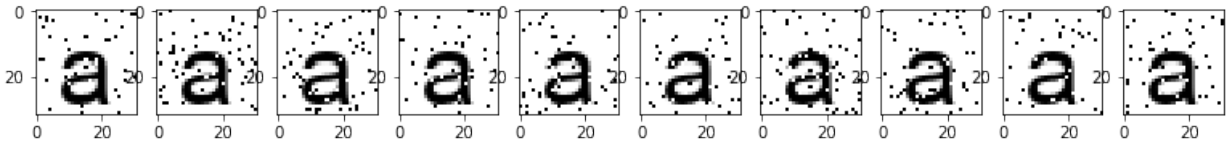
```
!! X correct=10 wrong= 0 answers=['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
```



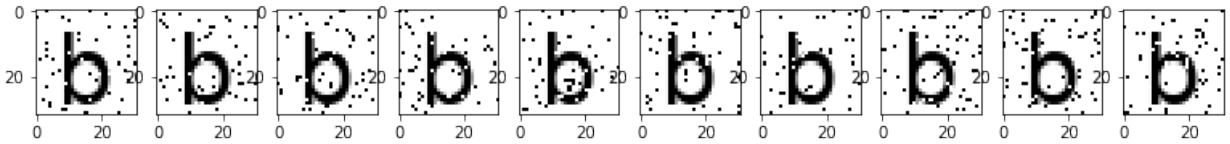
```
!! Y correct=10 wrong= 0 answers=['Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y']
```



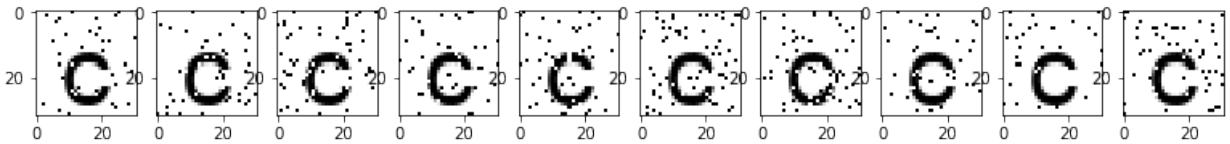
```
!! Z correct=10 wrong= 0 answers=['Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z']
```



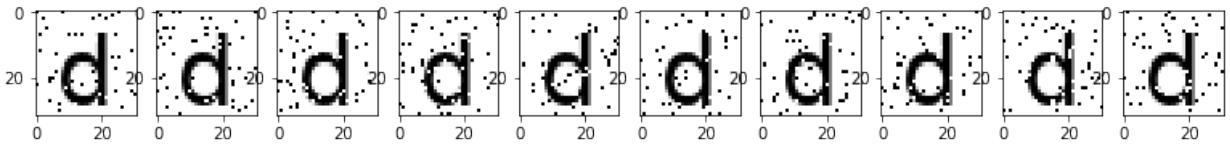
```
!! a correct=10 wrong= 0 answers=['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```



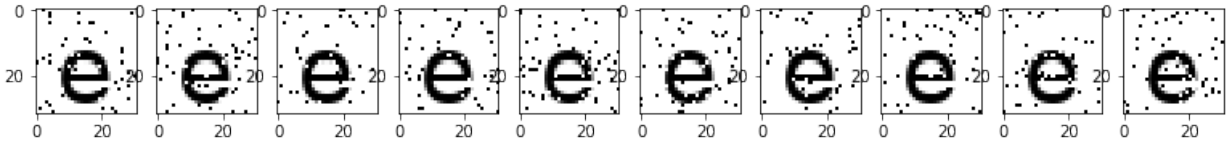
```
!! b correct= 7 wrong= 3 answers=['b', 'b', 'b', 'h', 'b', 'o', 'b', 'h', 'b', 'b']
```



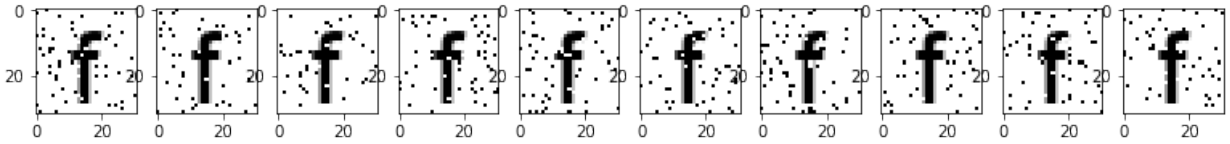
```
!! c correct=10 wrong= 0 answers=['c', 'c', 'c', 'c', 'c', 'c', 'c', 'c', 'c', 'c']
```



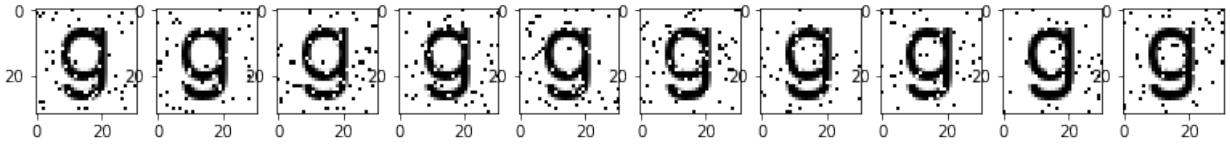
```
!! d correct=10 wrong= 0 answers=['d', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd']
```



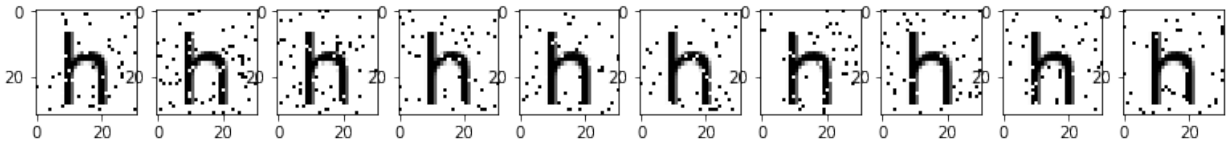
```
!! e correct= 9 wrong= 1 answers=['e', 'e', 'e', 'e', 'e', 'e', 'e', 'e', 'o', 'e']
```



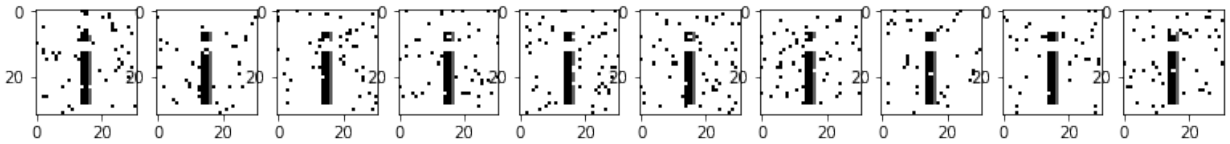
```
!! f correct= 6 wrong= 4 answers=['i', 'f', 'f', 'I', 'I', 'I', 'f', 'f', 'f', 'f']
```



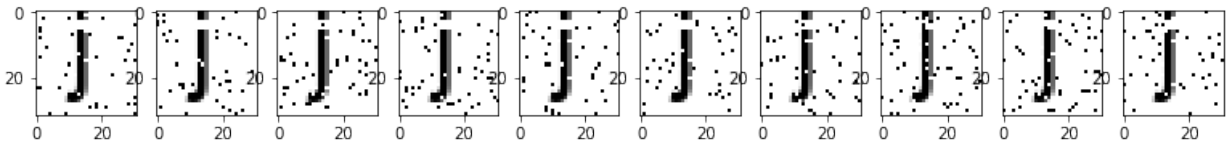
```
!! g correct=10 wrong= 0 answers=['g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g']
```



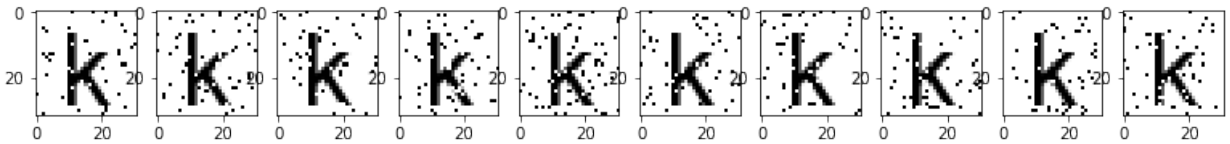
```
!! h correct=10 wrong= 0 answers=['h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h']
```



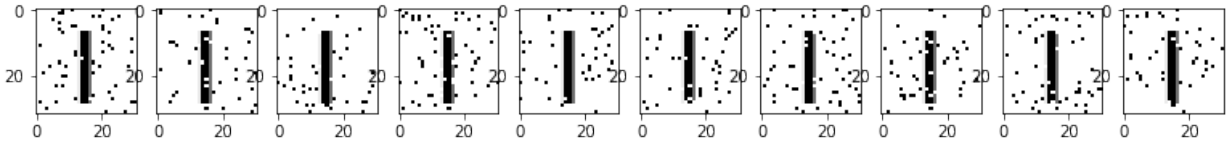
```
!! i correct=10 wrong= 0 answers=['i', 'i', 'i', 'i', 'i', 'i', 'i', 'i', 'i', 'i']
```



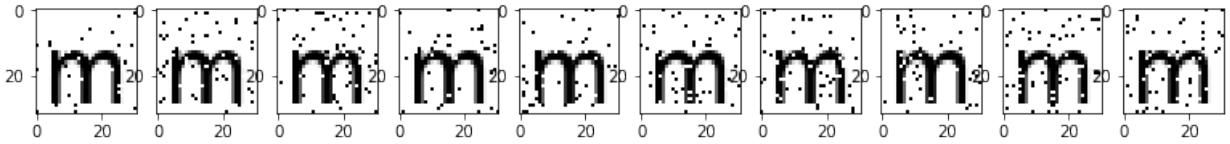
```
!! j correct=10 wrong= 0 answers=['j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j']
```



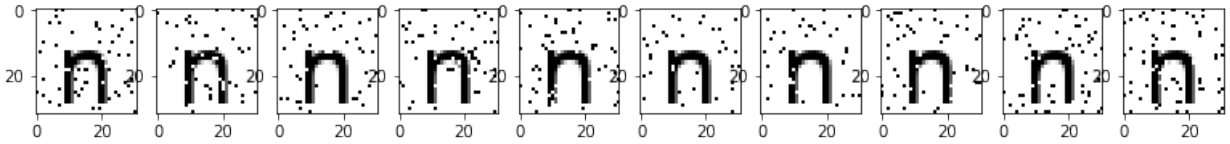
```
!! k correct=10 wrong= 0 answers=['k', 'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k']
```



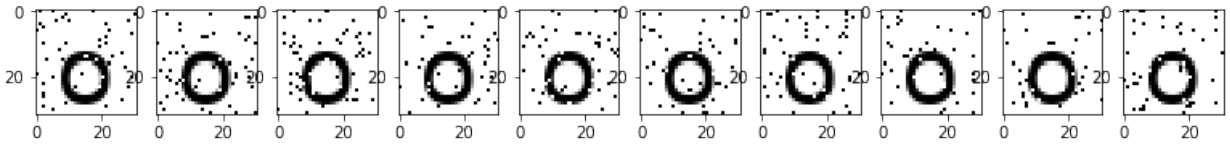
```
!! l correct= 0 wrong=10 answers=['i', 'i', 'i', 'i', 'i', 'i', 'i', 'i', 'i', 'i']
```



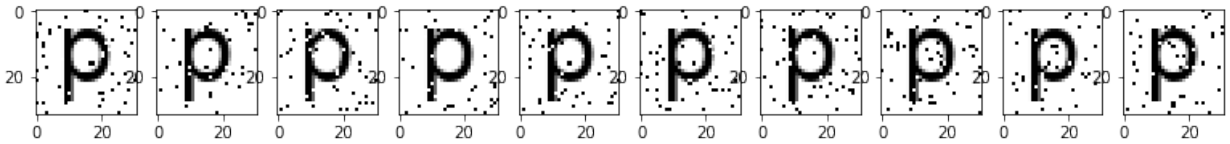
```
!! m correct=10 wrong= 0 answers=['m', 'm', 'm', 'm', 'm', 'm', 'm', 'm', 'm', 'm']
```



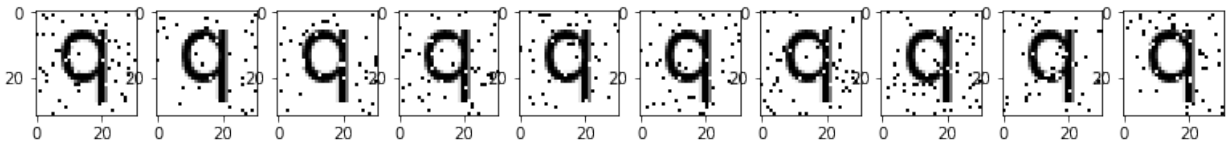
```
!! n correct=10 wrong= 0 answers=['n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n']
```



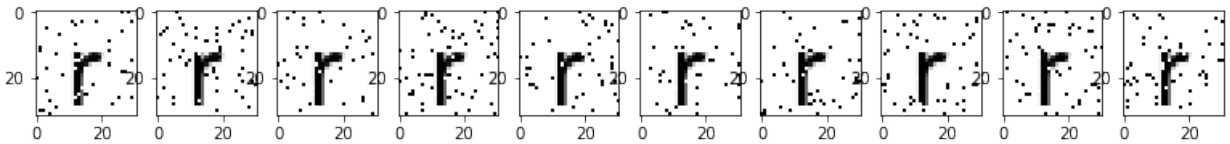
```
!! o correct=10 wrong= 0 answers=['o', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'o']
```



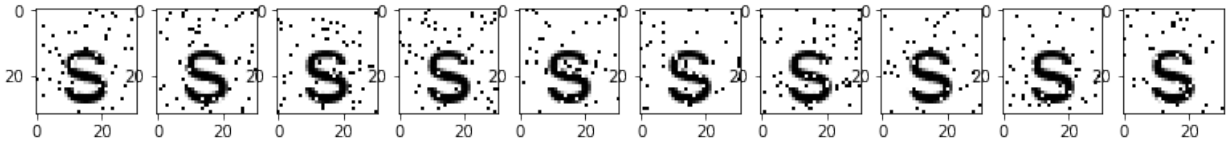
```
!! p correct=10 wrong= 0 answers=['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p']
```



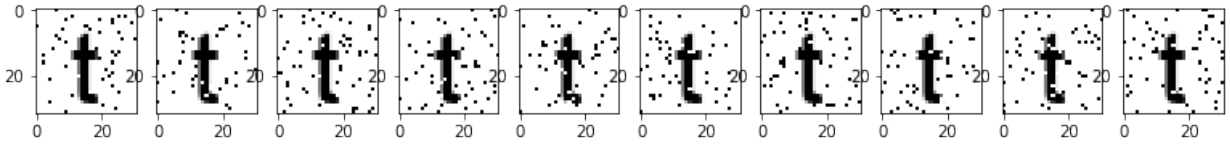
```
!! q correct=10 wrong= 0 answers=['q', 'q', 'q', 'q', 'q', 'q', 'q', 'q', 'q', 'q']
```



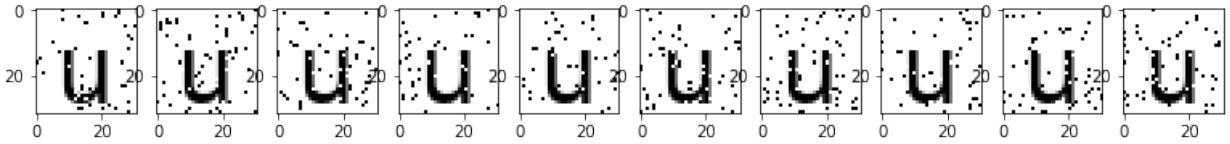
```
!! r correct=10 wrong= 0 answers=['r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r']
```



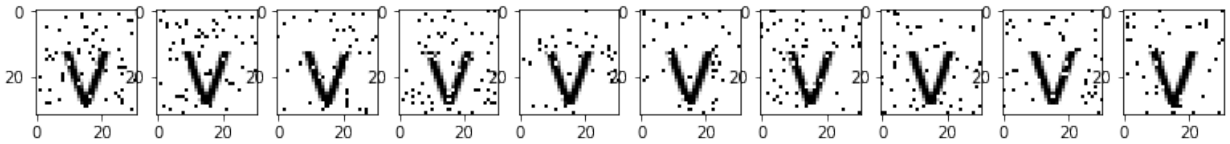
```
!! s correct=10 wrong= 0 answers=['s', 's', 's', 's', 's', 's', 's', 's', 's', 's']
```



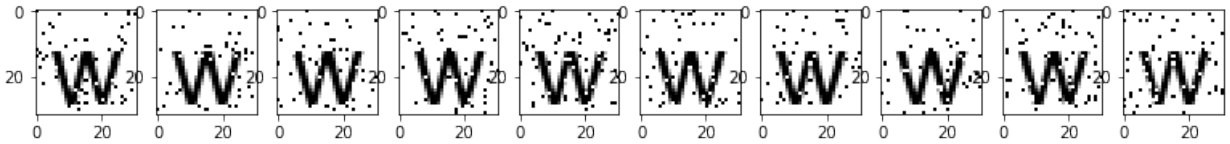
```
!! t correct= 9 wrong= 1 answers=['t', 't', 't', 't', 't', 't', 't', 'i', 't', 't']
```



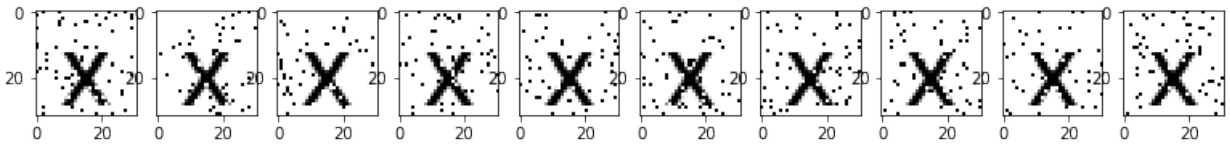
```
!! u correct=10 wrong= 0 answers=['u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u']
```



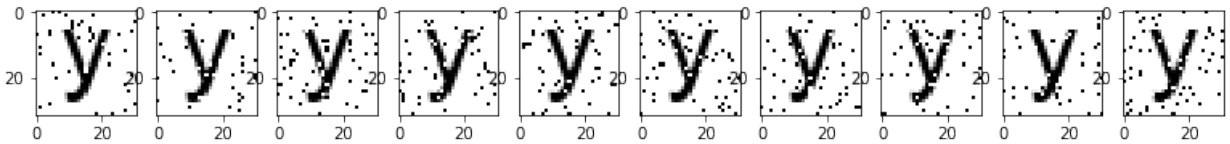
```
!! v correct=10 wrong= 0 answers=['v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v']
```



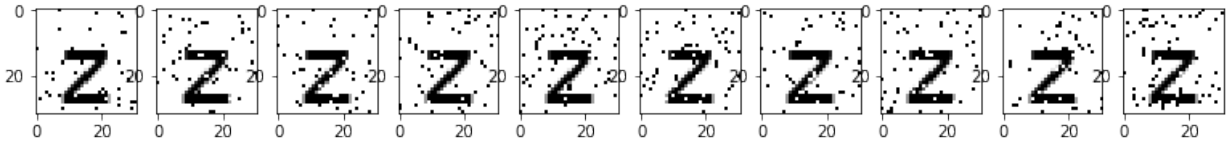
```
!! w correct=10 wrong= 0 answers=['w', 'w', 'w', 'w', 'w', 'w', 'w', 'w', 'w', 'w']
```



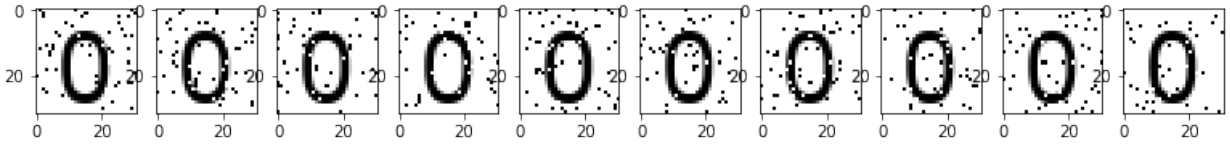
```
!! x correct=10 wrong= 0 answers=['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']
```



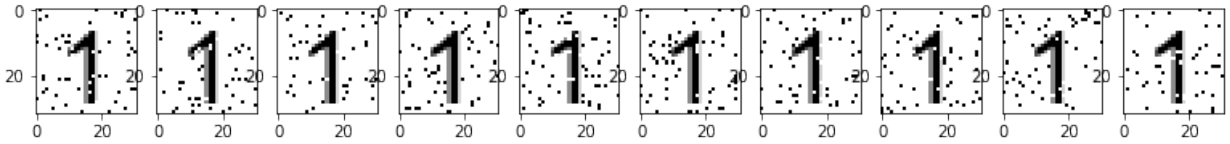
```
!! y correct=10 wrong= 0 answers=['y', 'y', 'y', 'y', 'y', 'y', 'y', 'y', 'y', 'y']
```



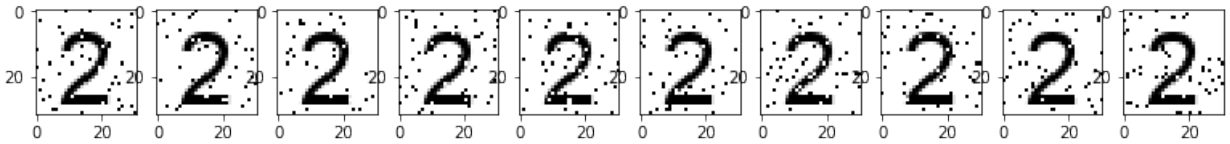
```
!! z correct=10 wrong= 0 answers=['z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z']
```



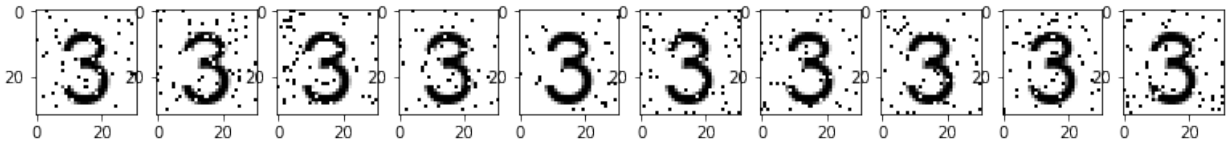
```
!! 0 correct=10 wrong= 0 answers=['0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
```



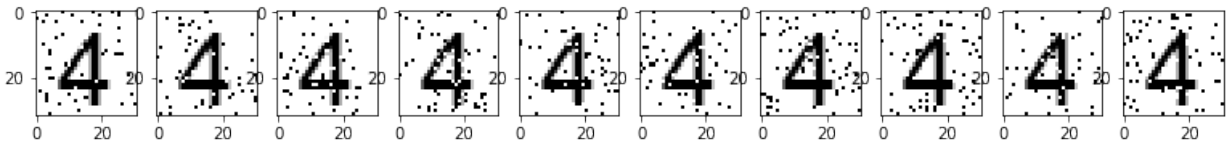
```
!! 1 correct=10 wrong= 0 answers=['1', '1', '1', '1', '1', '1', '1', '1', '1', '1']
```



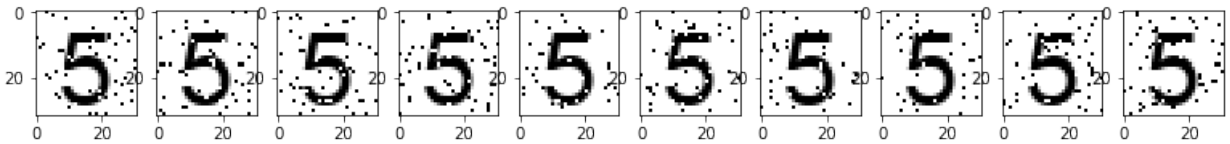
```
!! 2 correct=10 wrong= 0 answers=['2', '2', '2', '2', '2', '2', '2', '2', '2', '2']
```



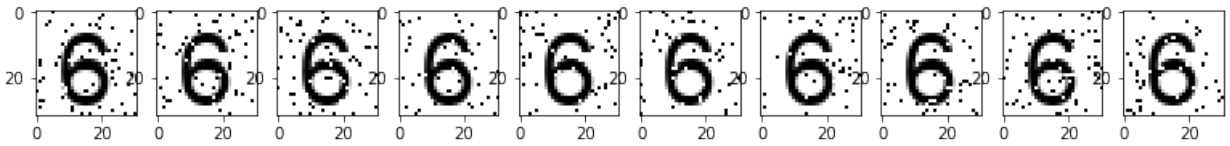
```
!! 3 correct=10 wrong= 0 answers=['3', '3', '3', '3', '3', '3', '3', '3', '3', '3']
```



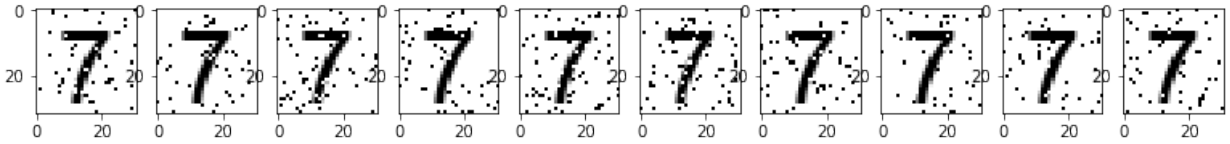
```
!! 4 correct=10 wrong= 0 answers=['4', '4', '4', '4', '4', '4', '4', '4', '4', '4']
```



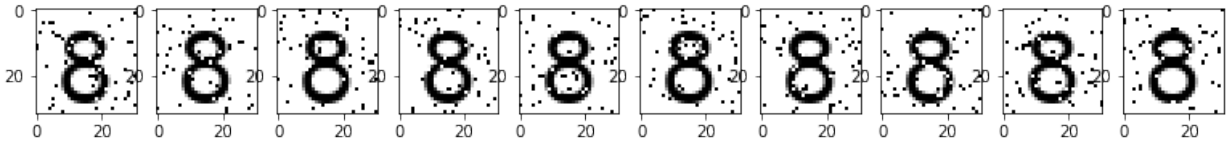
```
!! 5 correct=10 wrong= 0 answers=['5', '5', '5', '5', '5', '5', '5', '5', '5', '5']
```



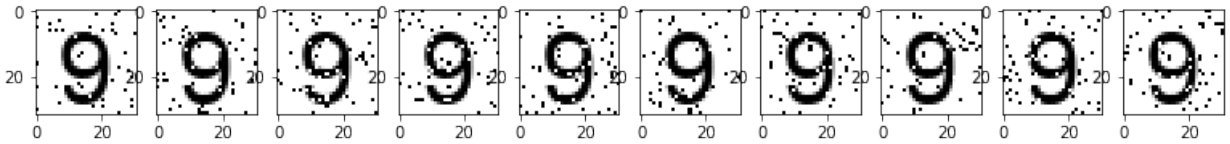
```
!! 6 correct=10 wrong= 0 answers=['6', '6', '6', '6', '6', '6', '6', '6', '6', '6']
```



```
!! 7 correct=10 wrong= 0 answers=['7', '7', '7', '7', '7', '7', '7', '7', '7', '7']
```



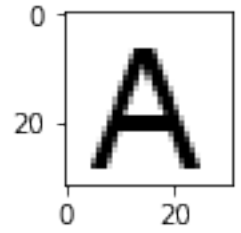
```
!! 8 correct= 9 wrong= 1 answers=['8', '6', '8', '8', '8', '8', '8', '8', '8', '8']
```



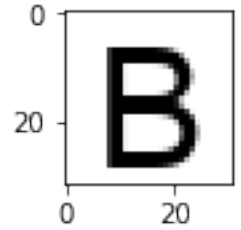
```
!! 9 correct= 7 wrong= 3 answers=['9', '9', '0', '9', '9', '9', '0', '0', '9', '9']
```

Testing with no noise

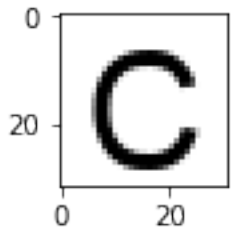
```
In [18]: for x in 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789':
             iter_read(x, p=0, n=1)
```



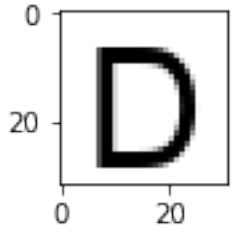
```
!! A correct= 1 wrong= 0 answers=['A']
```



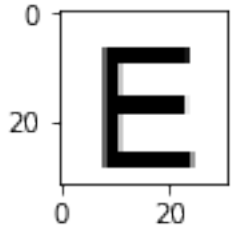
```
!! B correct= 1 wrong= 0 answers=['B']
```



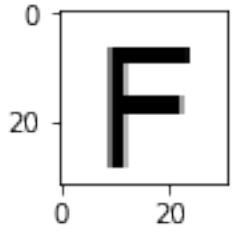

```
!! C correct= 1 wrong= 0 answers=['C']
```



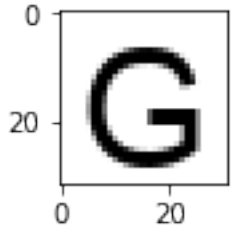

```
!! D correct= 1 wrong= 0 answers=['D']
```



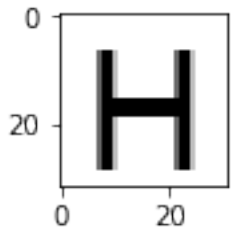

```
!! E correct= 1 wrong= 0 answers=['E']
```



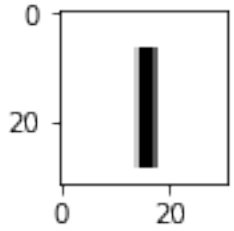

```
!! F correct= 1 wrong= 0 answers=['F']
```




```
!! G correct= 1 wrong= 0 answers=['G']
```

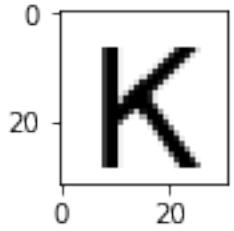



```
!! H correct= 1 wrong= 0 answers=['H']
```

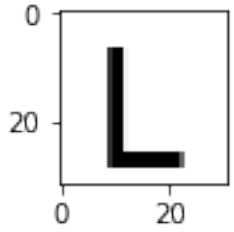



```
!! I correct= 1 wrong= 0 answers=['I']
```

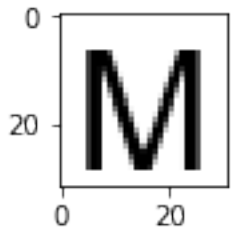

```
!! J correct= 1 wrong= 0 answers=['J']
```



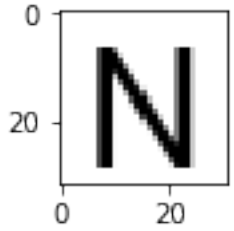

```
!! K correct= 1 wrong= 0 answers=['K']
```



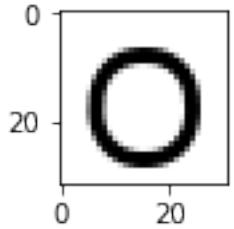

```
!! L correct= 1 wrong= 0 answers=['L']
```



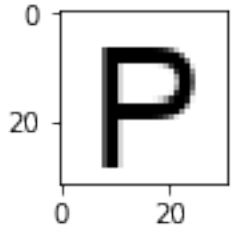

```
!! M correct= 1 wrong= 0 answers=['M']
```



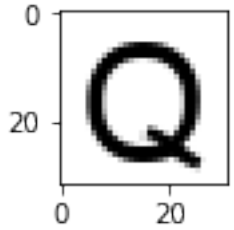

```
!! N correct= 1 wrong= 0 answers=['N']
```



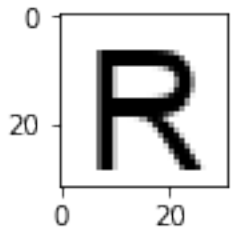

```
!! O correct= 1 wrong= 0 answers=['O']
```



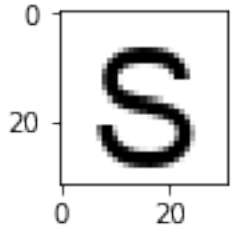

```
!! P correct= 1 wrong= 0 answers=['P']
```



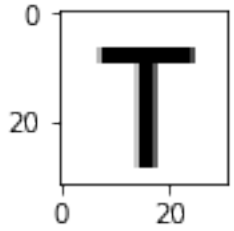

```
!! Q correct= 1 wrong= 0 answers=['Q']
```



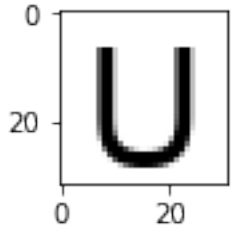

```
!! R correct= 1 wrong= 0 answers=['R']
```



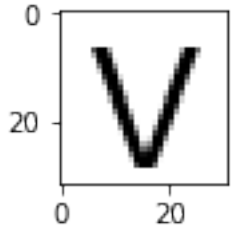

```
!! S correct= 1 wrong= 0 answers=['S']
```



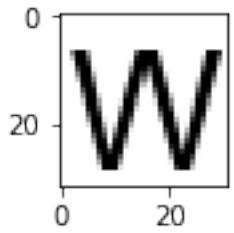

```
!! T correct= 1 wrong= 0 answers=['T']
```



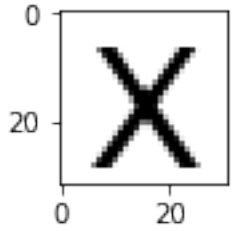

```
!! U correct= 1 wrong= 0 answers=['U']
```



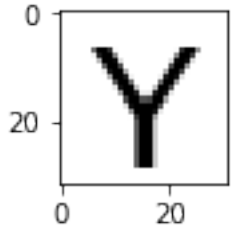

```
!! V correct= 1 wrong= 0 answers=['V']
```



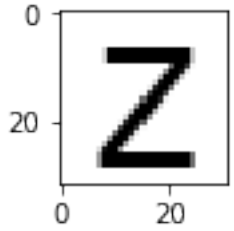

```
!! W correct= 1 wrong= 0 answers=['W']
```



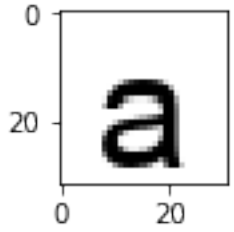

```
!! X correct= 1 wrong= 0 answers=['X']
```



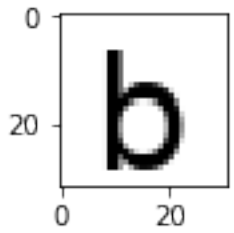

```
!! Y correct= 1 wrong= 0 answers=['Y']
```



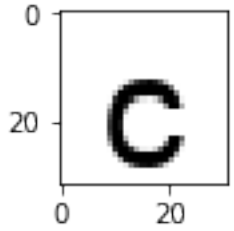

```
!! Z correct= 1 wrong= 0 answers=['Z']
```



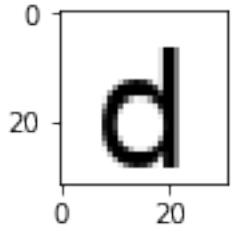

```
!! a correct= 1 wrong= 0 answers=['a']
```



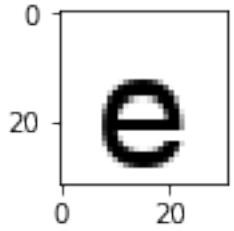

```
!! b correct= 1 wrong= 0 answers=['b']
```



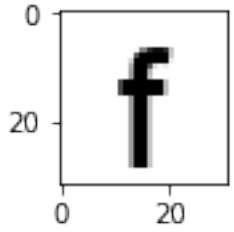

```
!! c correct= 1 wrong= 0 answers=['c']
```



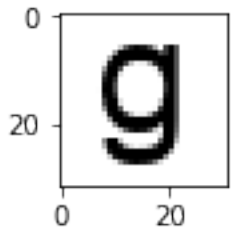

```
!! d correct= 1 wrong= 0 answers=['d']
```



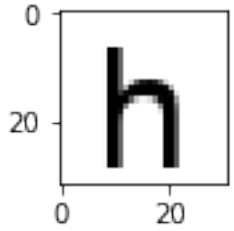

```
!! e correct= 1 wrong= 0 answers=['e']
```



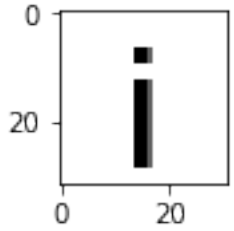

```
!! f correct= 1 wrong= 0 answers=['f']
```



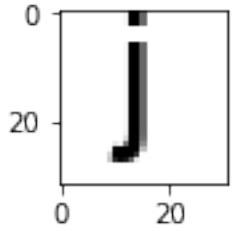

```
!! g correct= 1 wrong= 0 answers=['g']
```



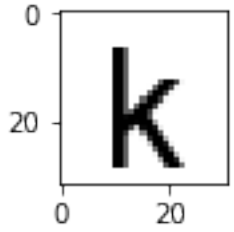

```
!! h correct= 1 wrong= 0 answers=['h']
```



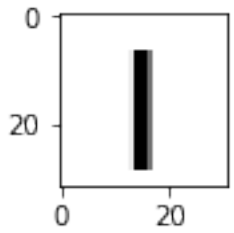

```
!! i correct= 1 wrong= 0 answers=['i']
```



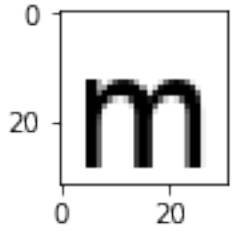

```
!! j correct= 1 wrong= 0 answers=['j']
```



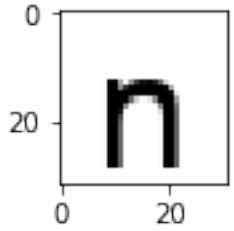

```
!! k correct= 1 wrong= 0 answers=['k']
```



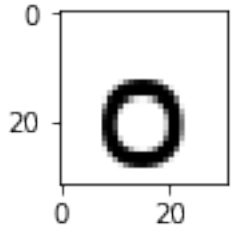

```
!! l correct= 0 wrong= 1 answers=['i']
```



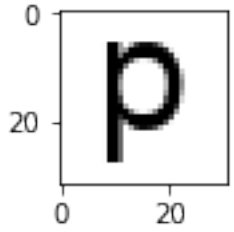

```
!! m correct= 1 wrong= 0 answers=['m']
```



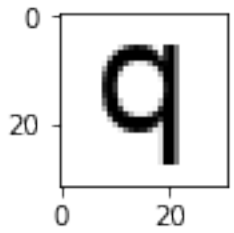

```
!! n correct= 1 wrong= 0 answers=['n']
```



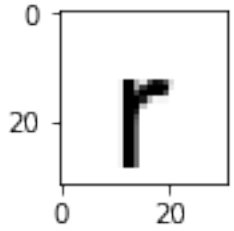

```
!! o correct= 1 wrong= 0 answers=['o']
```



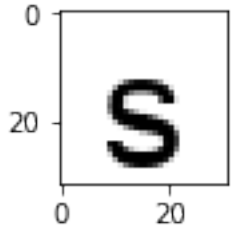

```
!! p correct= 1 wrong= 0 answers=['p']
```



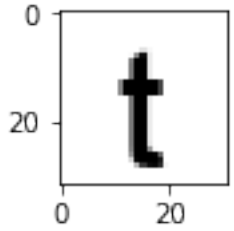

```
!! q correct= 1 wrong= 0 answers=['q']
```



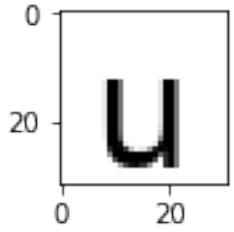

```
!! r correct= 1 wrong= 0 answers=['r']
```



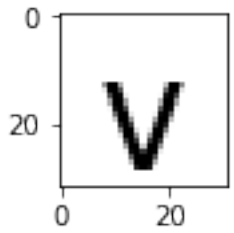

```
!! s correct= 1 wrong= 0 answers=['s']
```



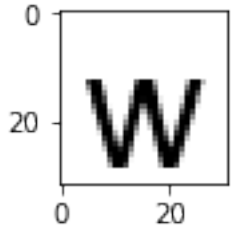

```
!! t correct= 1 wrong= 0 answers=['t']
```



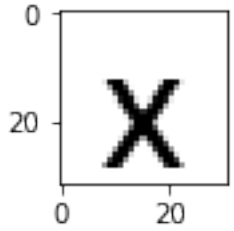

```
!! u correct= 1 wrong= 0 answers=['u']
```



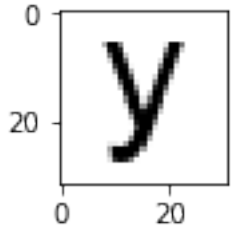

```
!! v correct= 1 wrong= 0 answers=['v']
```



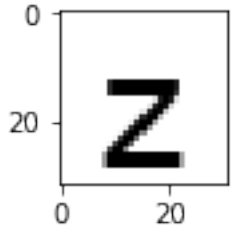

```
!! w correct= 1 wrong= 0 answers=['w']
```



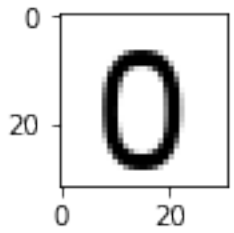

```
!! x correct= 1 wrong= 0 answers=['x']
```



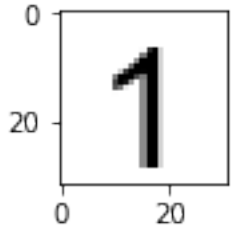

```
!! y correct= 1 wrong= 0 answers=['y']
```



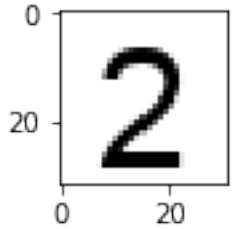

```
!! z correct= 1 wrong= 0 answers=['z']
```



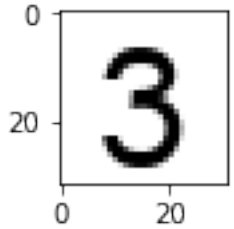

```
!! 0 correct= 1 wrong= 0 answers=['0']
```



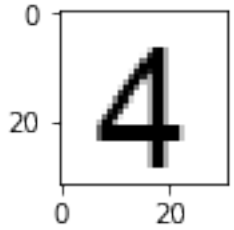

```
!! 1 correct= 1 wrong= 0 answers=['1']
```



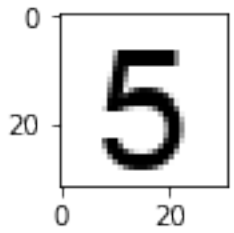

```
!! 2 correct= 1 wrong= 0 answers=['2']
```



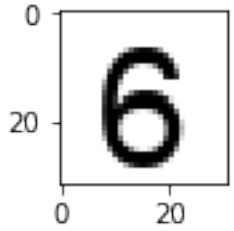

```
!! 3 correct= 1 wrong= 0 answers=['3']
```



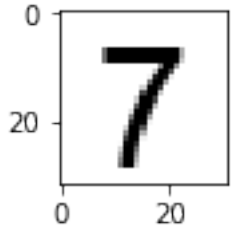

```
!! 4 correct= 1 wrong= 0 answers=['4']
```



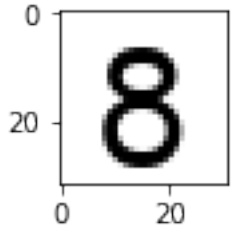

```
!! 5 correct= 1 wrong= 0 answers=['5']
```



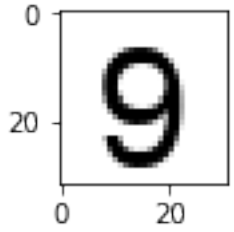

```
!! 6 correct= 1 wrong= 0 answers=['6']
```




```
!! 7 correct= 1 wrong= 0 answers=['7']
```




```
!! 8 correct= 1 wrong= 0 answers=['8']
```




```
!! 9 correct= 1 wrong= 0 answers=['9']
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index