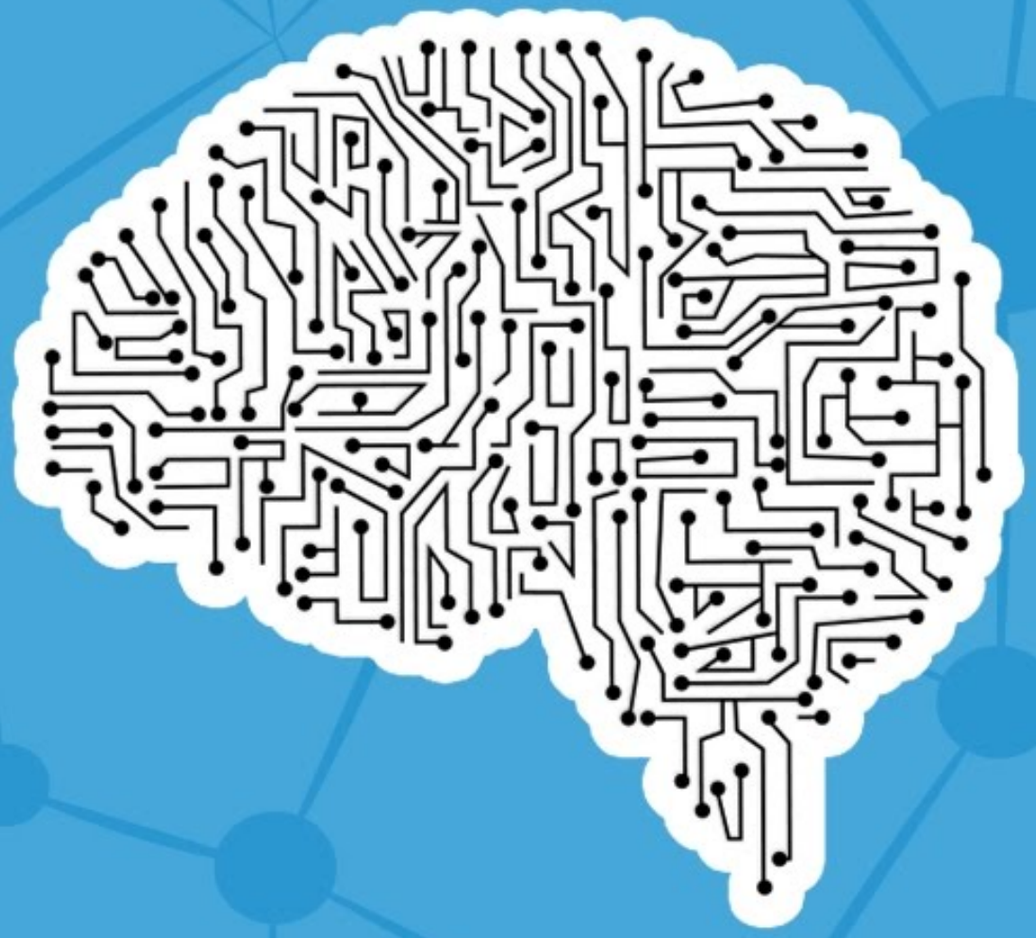
INTRODUCTION TO TENSORFLOW

for Deep Learning



SAPIENZA UNIVERSITÀ DI ROMA

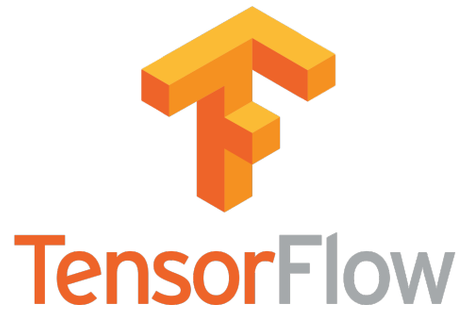
Mara Sorella
@maruscia

~ Web Algorithmics and Data Mining Lab

TensorFlow

TensorFlow™ is an open source software library for numerical computation using **data flow graphs**.

Originally developed (for internal use) by researchers and engineers working on the **Google Brain Team** for the purposes of conducting **machine learning** and **deep neural networks research.**



It was released under the Apache 2.0 open source license on November 2015.

What is a Tensor?

Tensor

a typed multi-dimensional array.

i.e. *a mini-batch of images*

a 4-D array of **floating point numbers** with dimensions `[batch_size, height, width, channels]`

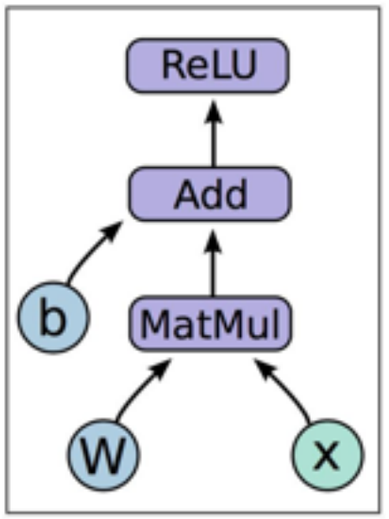
Operation

takes zero or more Tensors, performs some computation, and produces zero or more Tensors

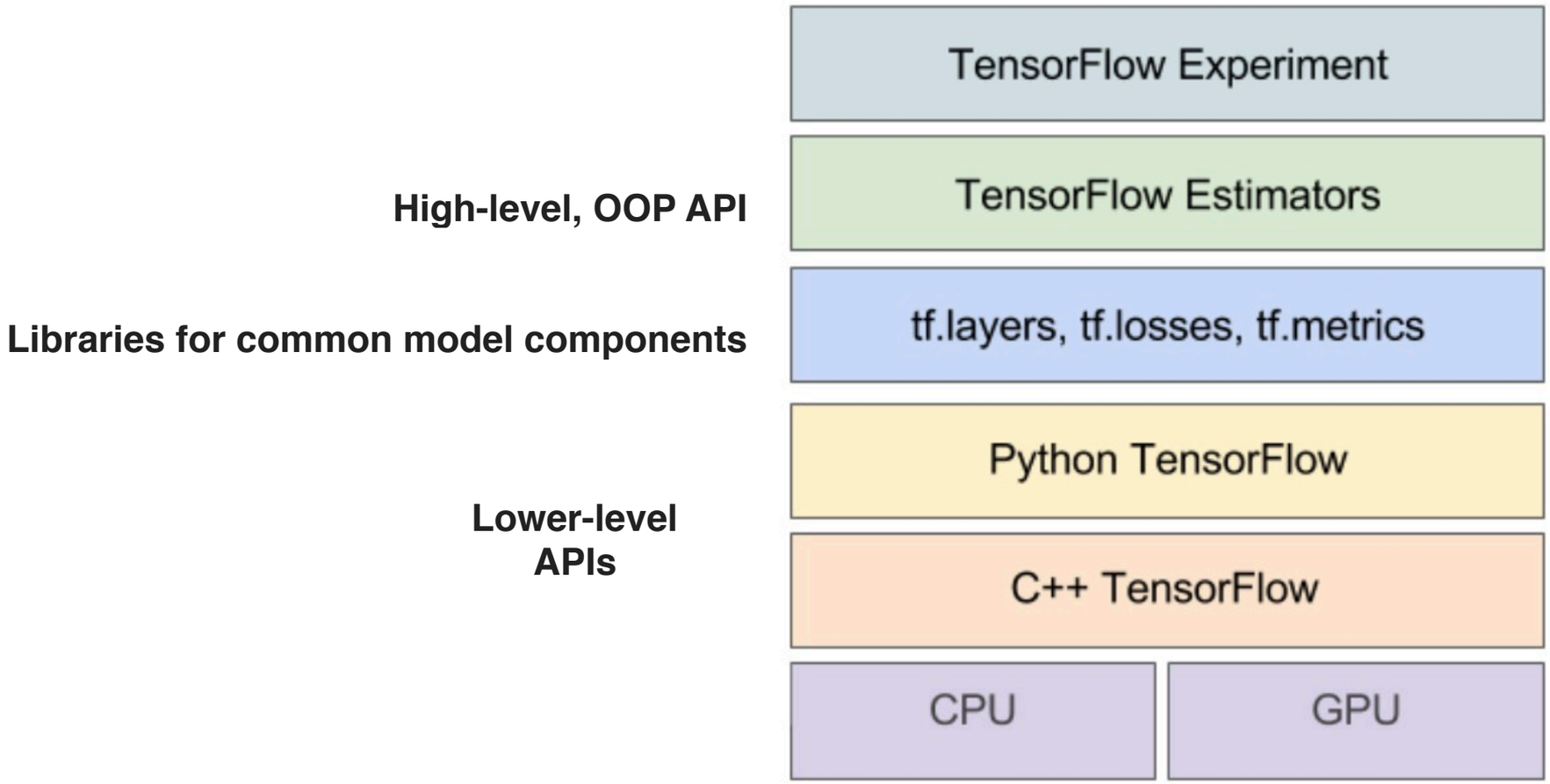
$$h = ReLU(Wx + b)$$

Data flow graph

- **Nodes** in the graph represent **operations**
- **Edges** are directed and represent **passing the result** of an operation (a tensor) as an operand to another operation



TensorFlow architecture



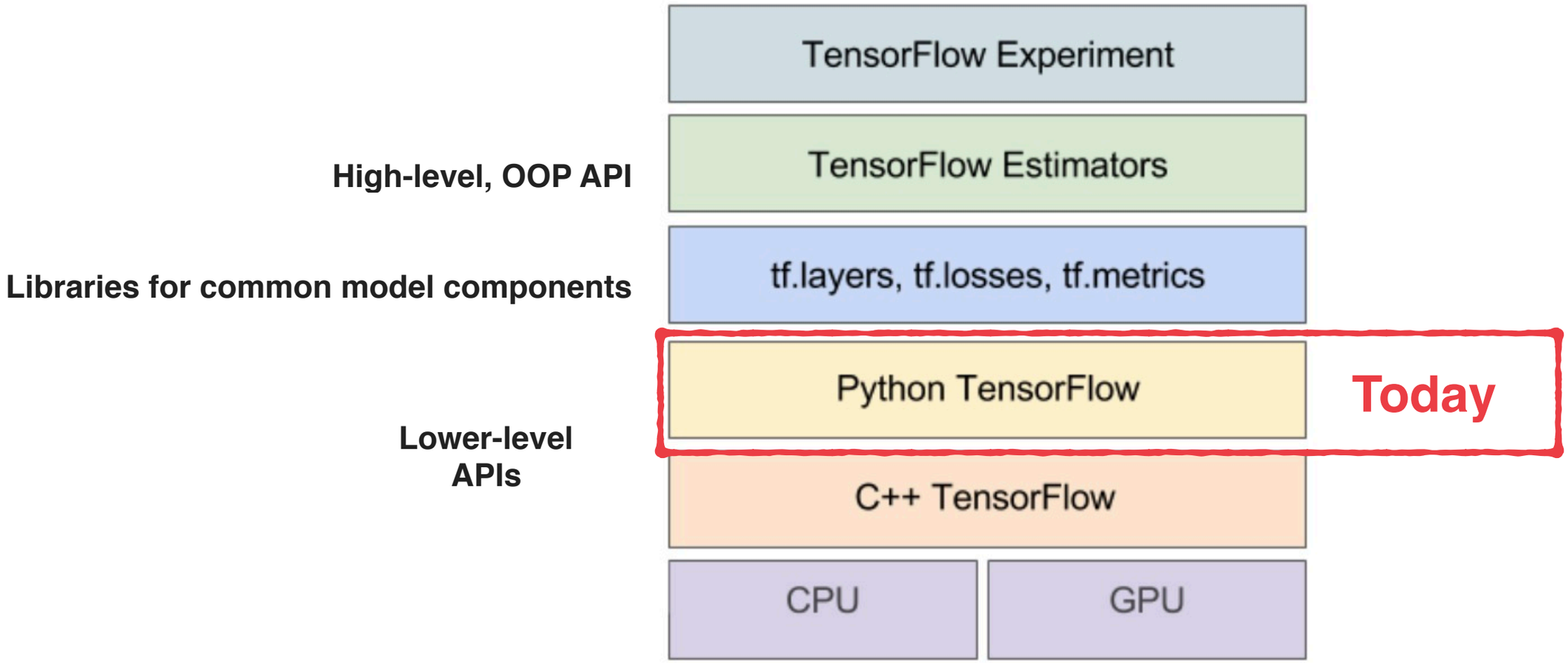
TensorFlow consists of the following two components:

- a graph prototype

  a **computation specification**.

- a runtime that executes the (distributed) graph

TensorFlow architecture



TensorFlow consists of the following two components:

- a graph prototype

  a **computation specification**.

- a runtime that executes the (distributed) graph

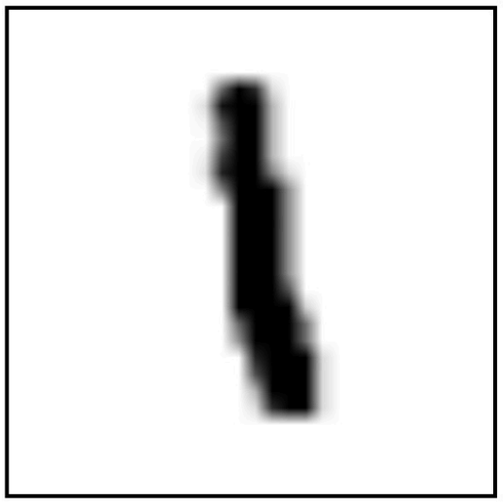
MNIST: Handwritten digits recognition

Righteously identified as the `Hello World` of machine learning.

- 60K labeled examples

label: 5 label: 0 label: 4 label: 1

- 28x28 pixel grayscale image



$$\mathbb{R}\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .6 & .8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .5 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .9 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .3 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

- 1-ch intensity matrix

A case of multilabel classification

We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

We want to be able to look at an image and know the probabilities for it being each digit.

A case of multilabel classification

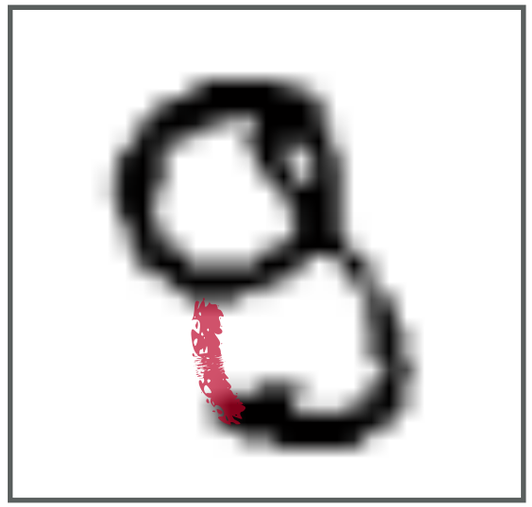
We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

We want to be able to look at an image and know the probabilities for it being each digit.

A case of multilabel classification

We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

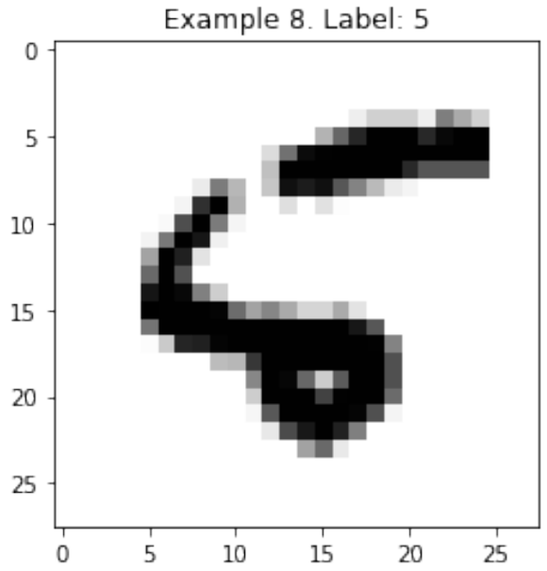
We want to be able to look at an image and know the probabilities for it being each digit.



A case of multilabel classification

We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

We want to be able to look at an image and know the probabilities for it being each digit.

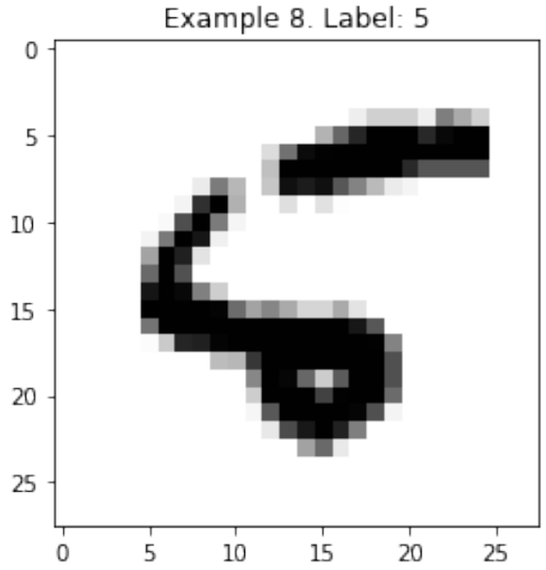


0	1.	….	5	6	….	9
0.02	0.01		0.5	0.45		0.01

A case of multilabel classification

We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

We want to be able to look at an image and know the probabilities for it being each digit.



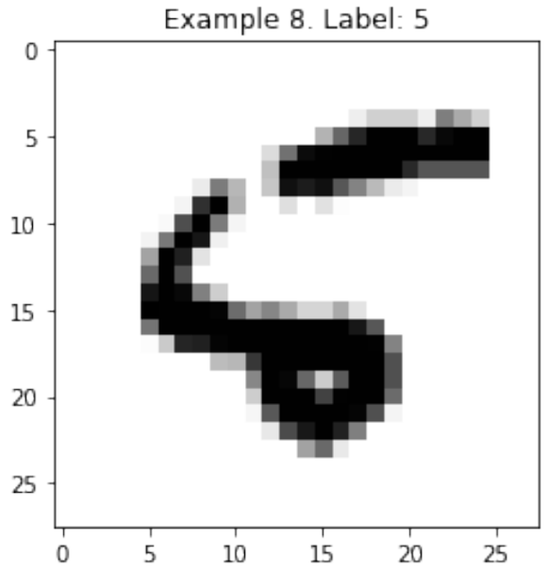
0	1.	….	5	6	….	9
0.02	0.01		0.5	0.45		0.01

unwrap

A case of multilabel classification

We know that every image in MNIST is a handwritten digit between zero and nine: **10 classes**

We want to be able to look at an image and know the probabilities for it being each digit.



0	1.	….	5	6	….	9
0.02	0.01		0.5	0.45		0.01

unwrap

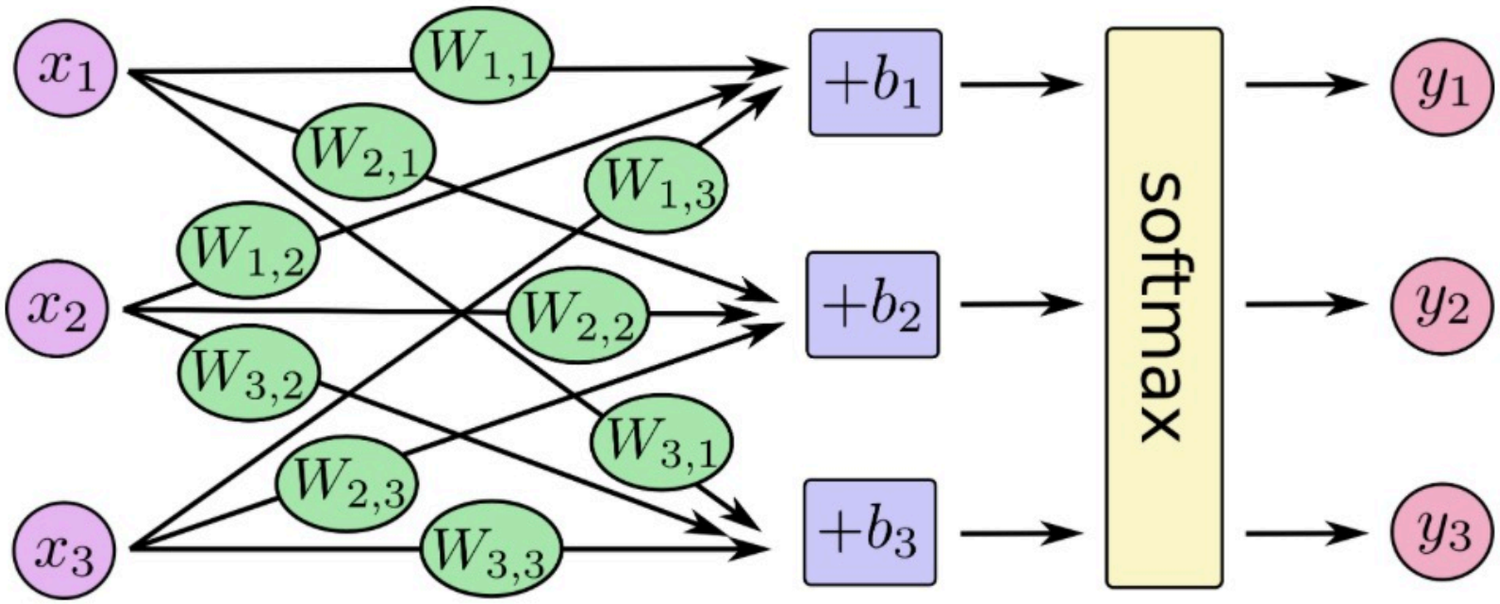
Evidence for a class i

$$\mathrm{L}_i = \sum_j W_{i,j} x_j + b_i$$

weight — *one per pixel*

bias — *one per class*

Softmax regression for MLC

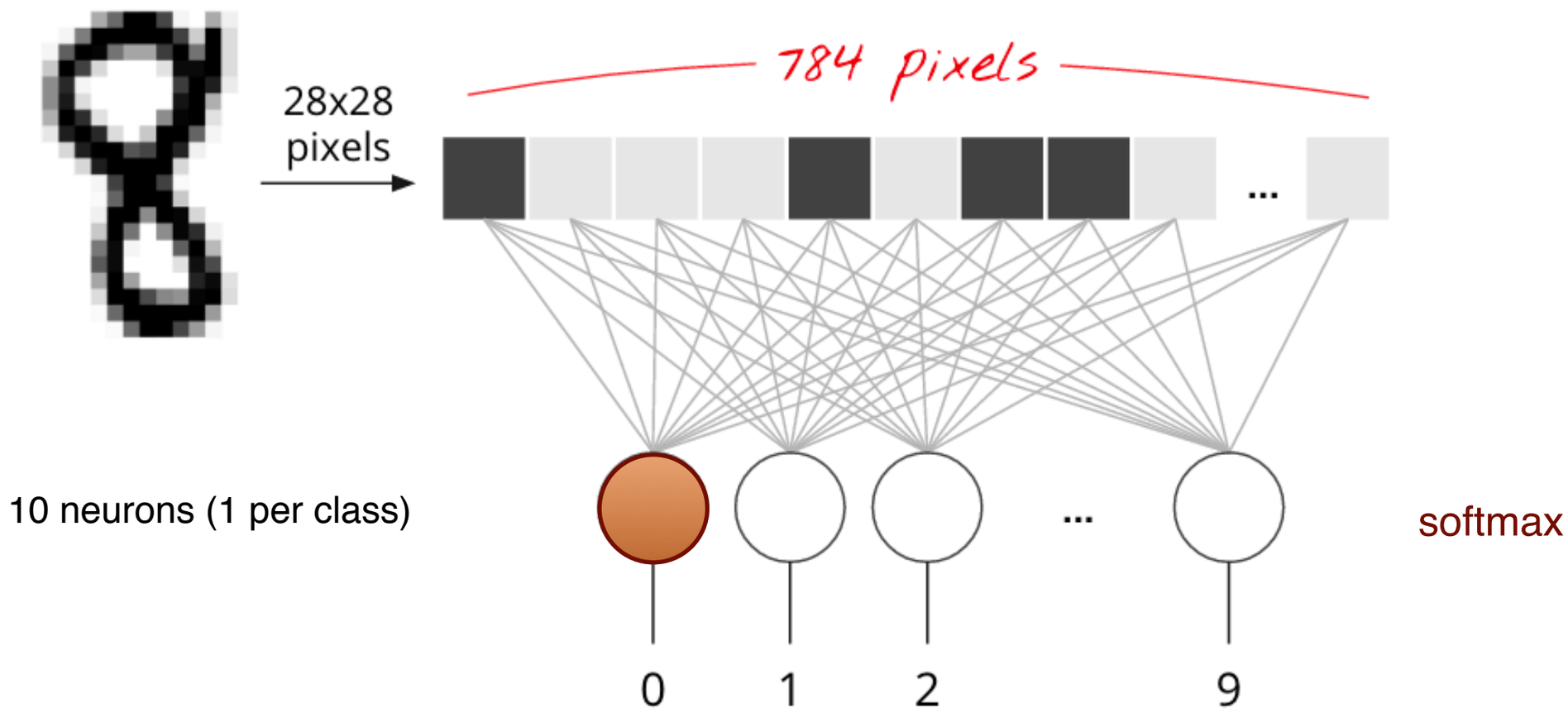


Probability distribution of class belonging

$$\mathrm{L}_i = \sum_j W_{i,j} x_j + b_i \qquad softmax(L_i) = \frac{e^{L_i}}{\|e^{L}\|}$$

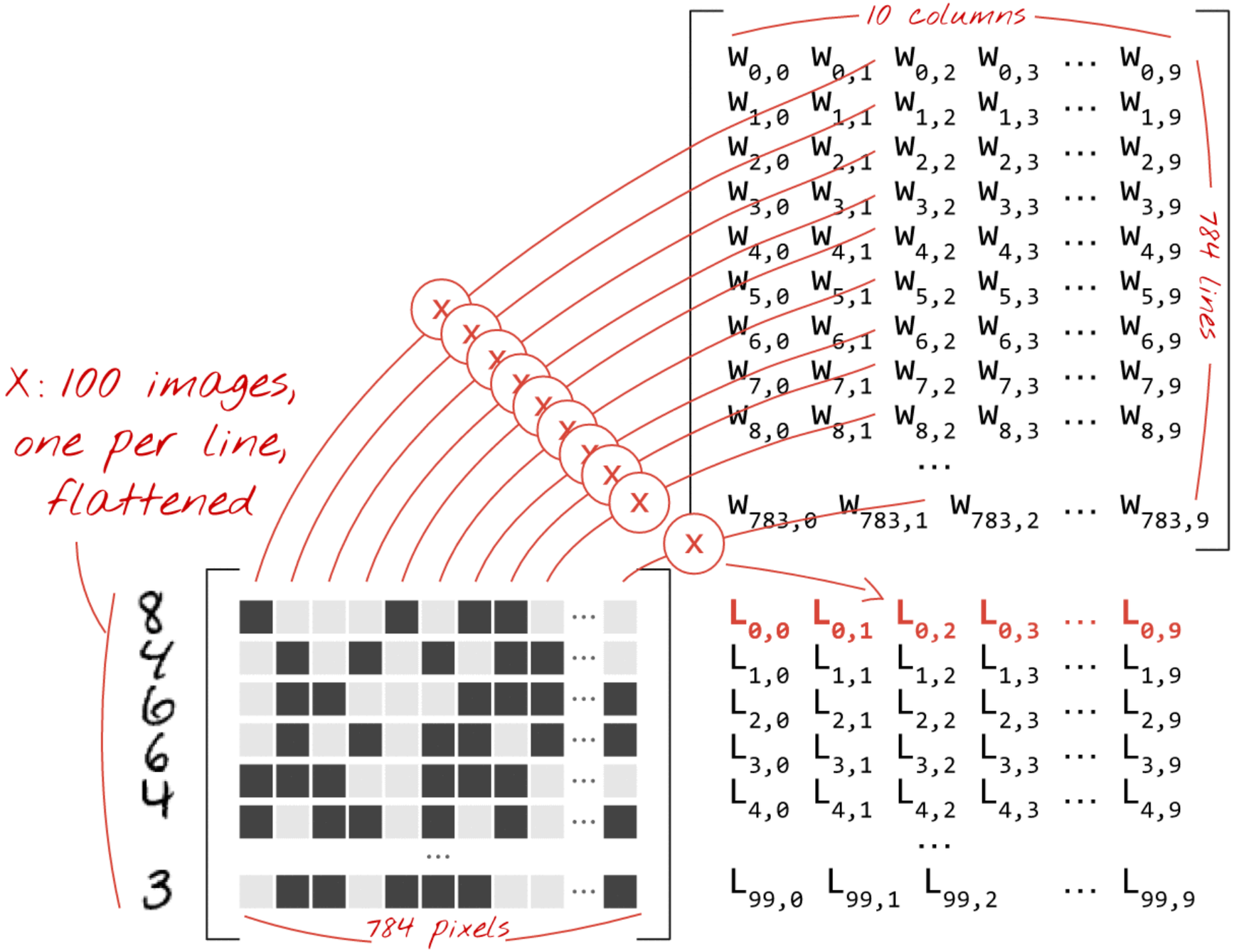
Our promise: build a neural network that achieves 99% accuracy in this task

1-Layer simple NN for MLC



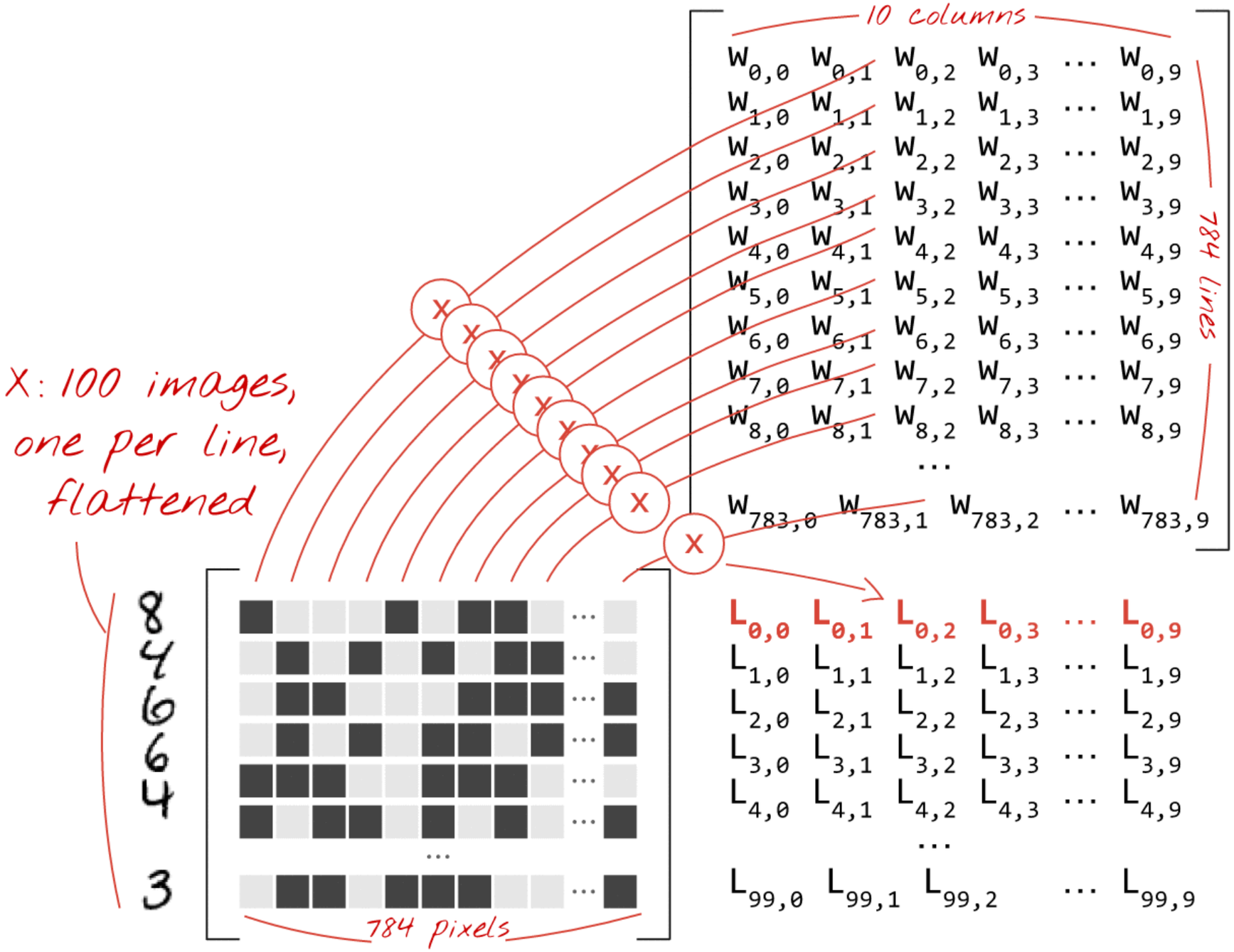
Activation function: a function (in this case softmax, yet also **sigmoid, ReLu,…**) that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer.

Working on a batch



$$L = X.W + b$$

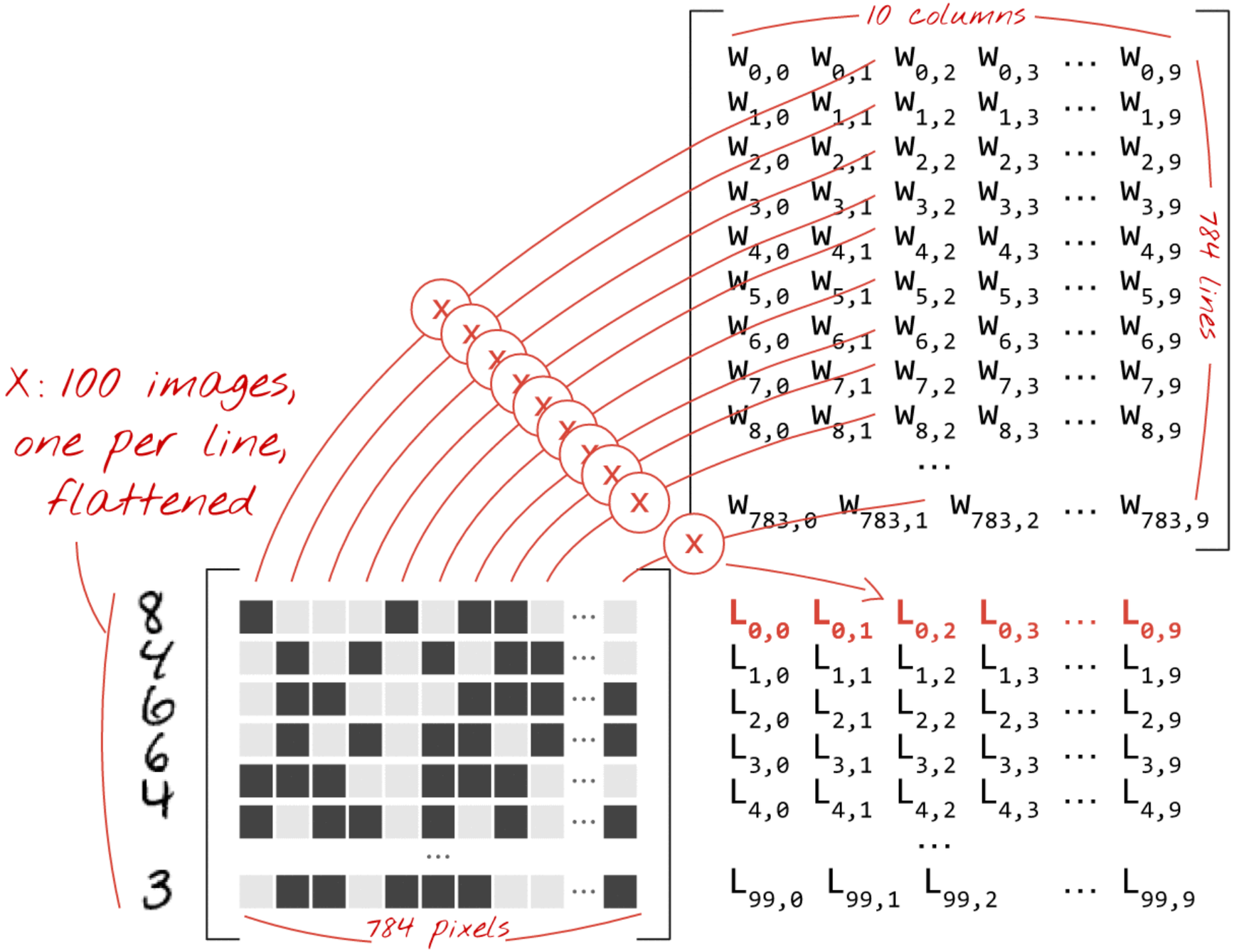
Working on a batch



[100,10] [1,10]

$$L = X.W + b$$

Working on a batch



[100,10] [1,10]

$$L = X.W + b$$

broadcast

Softmax over a batch of images

Predictions
Y[100, 10]

Images
X[100, 784]

Weights
W[784,10]

Biases
b[10]

$$Y = softmax(X.W + b)$$

applied line by line

matrix multiply

broadcast on all lines

tensor shapes in [ ]

...in TensorFlow (Python)

tensor shapes: X[100, 784] W[748,10] b[10]

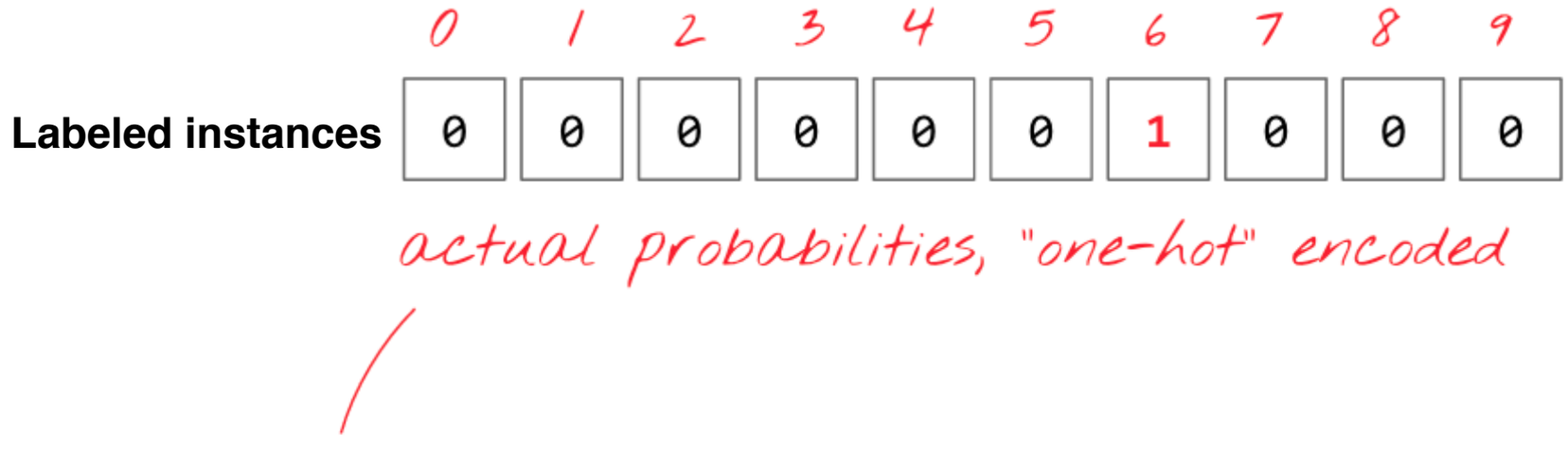
```
Y = tf.nn.softmax(tf.matmul(X, W) + b)
```

matrix multiply

broadcast on all lines

A loss function for MLC

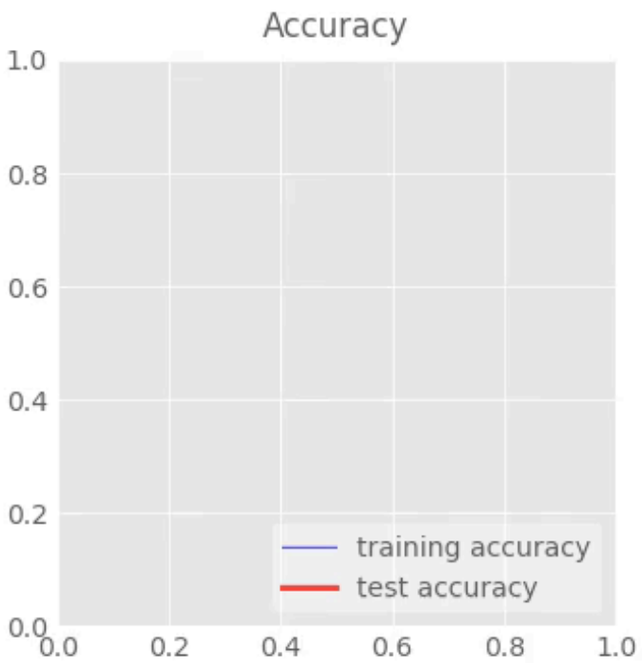
- **MNIST**: 60K images **train**, 10K **test**

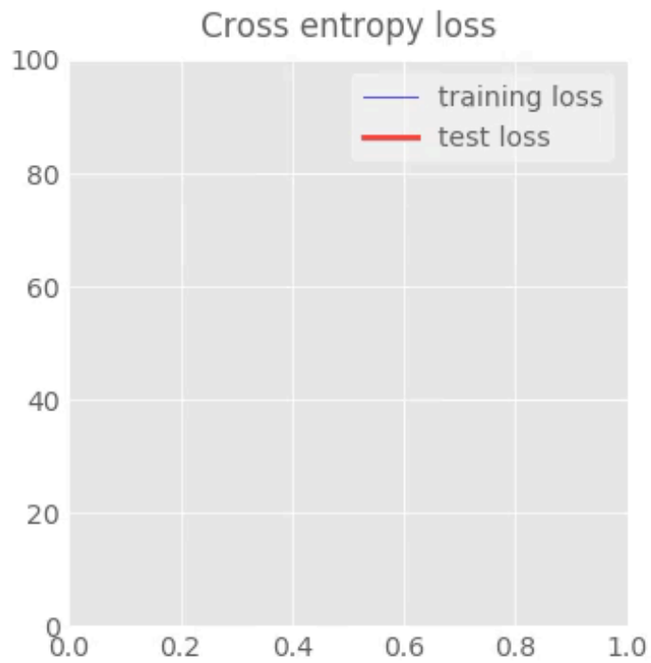


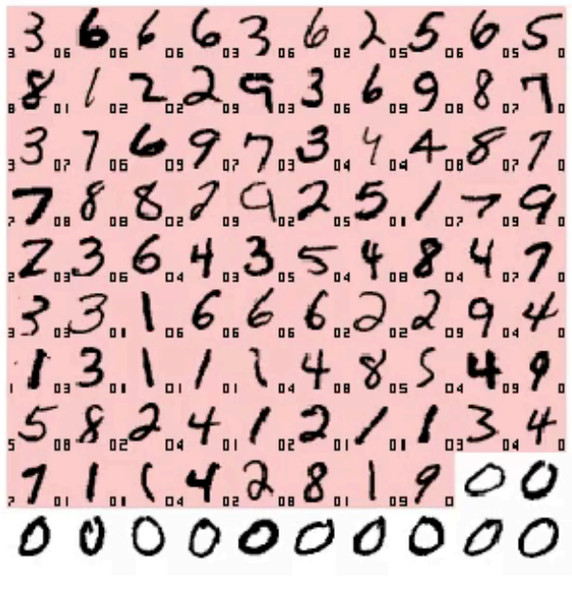
Training

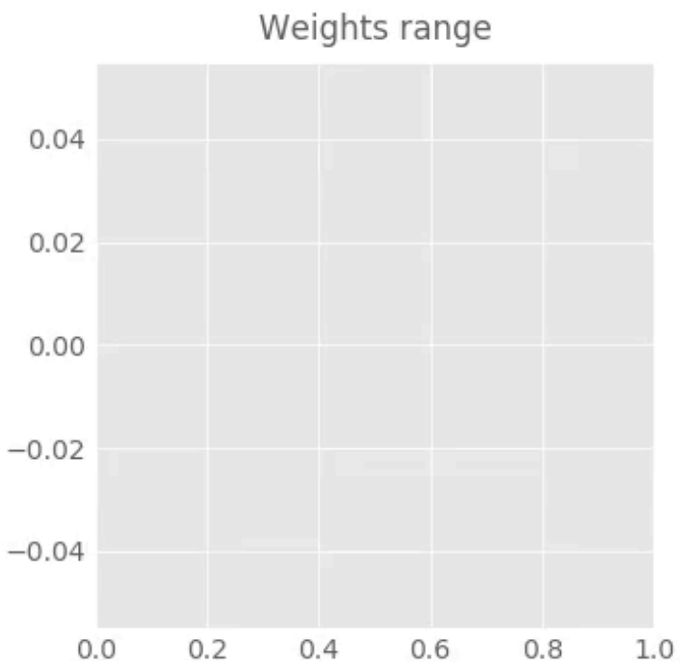
minimize CE, across
all training examples

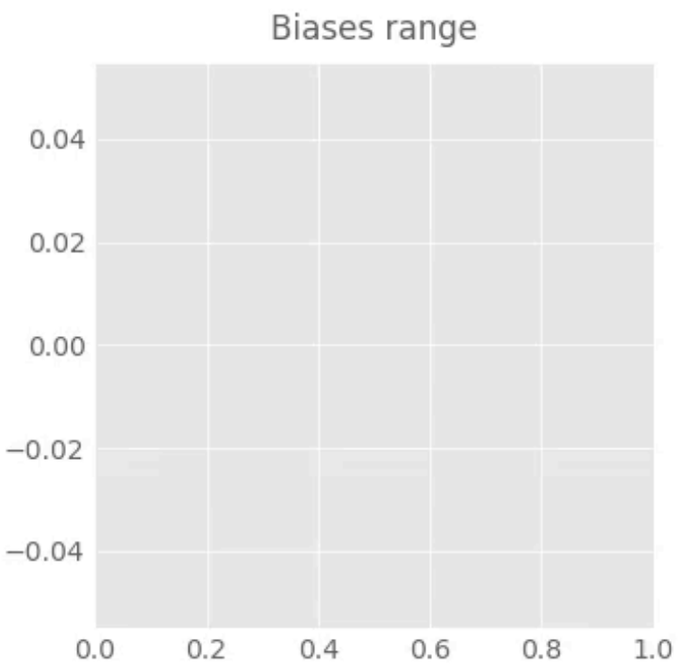
Demo: mnist_1.0_softmax.py



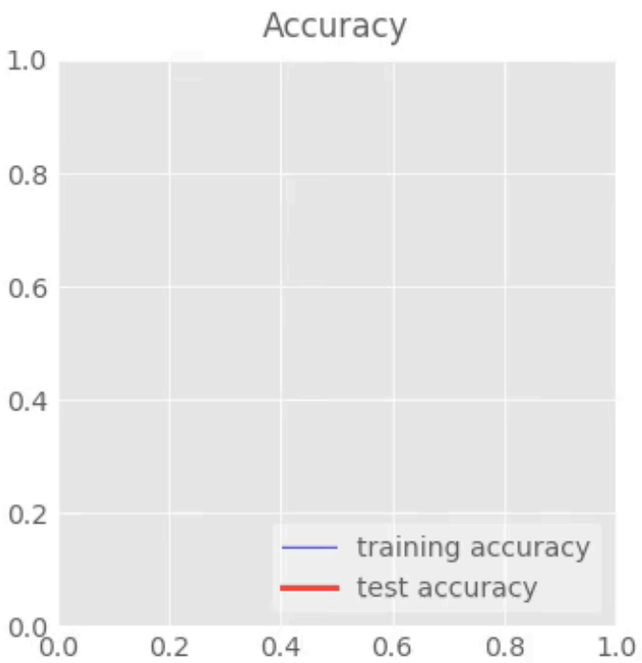


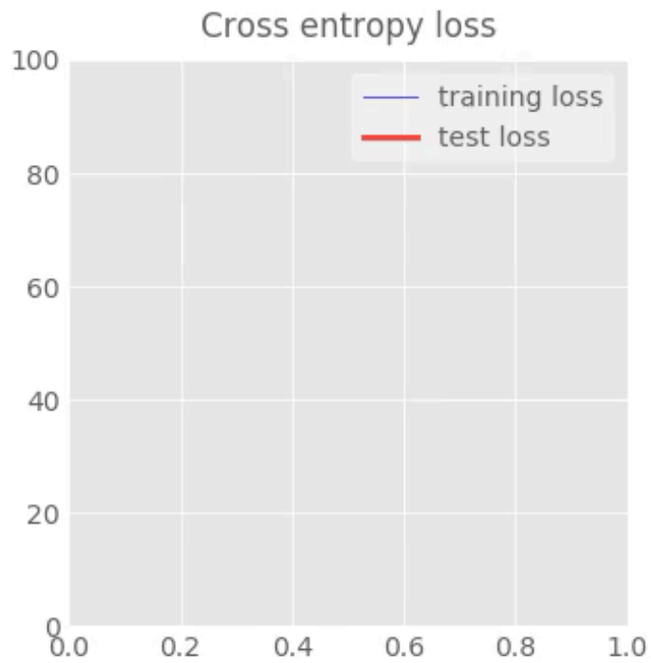


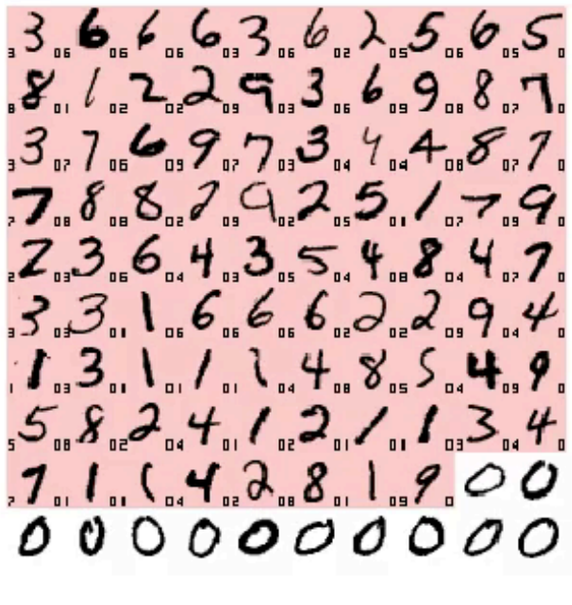


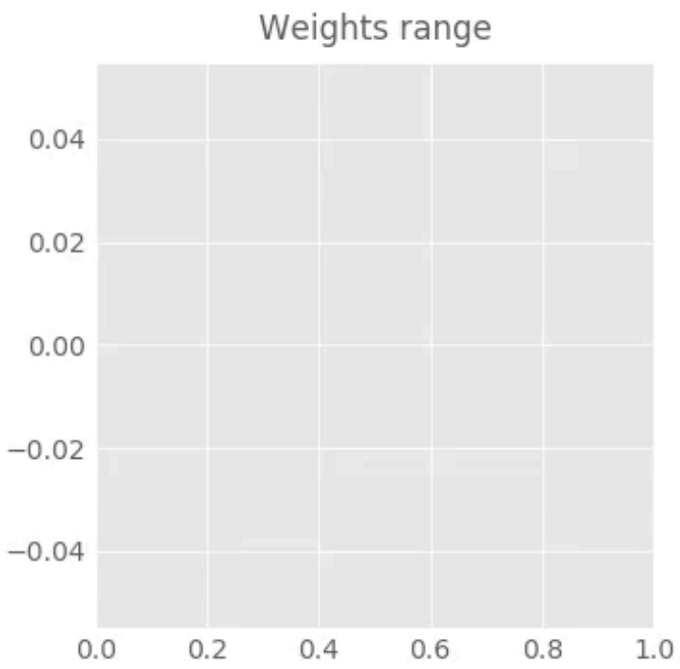


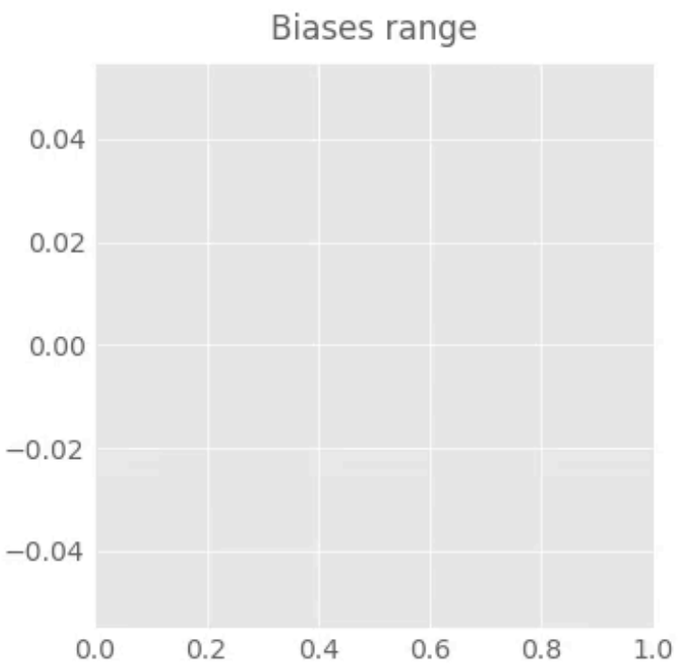
Demo: mnist_1.0_softmax.py











TensorFlow: init

Low level TF APIs

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

init = tf.initialize_all_variables()
```

this will become the batch size, 100

28 x 28 grayscale images

Training = computing variables W and b

Model

flattening images

```
# model
Y = tf.nn.softmax(tf.matmul(tf.reshape(X, [-1, 784]), W) + b)
# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])
```

"one-hot" encoded

```
# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))
```

"one-hot" decoding

```
# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
```

Optimizer

learning rate

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)
```

loss function

TF: full python code

initialisation

```
import tensorflow as tf
```

```
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
init = tf.initialize_all_variables()
```

model

```
# model
Y=tf.nn.softmax(tf.matmul(tf.reshape(X,[-1, 784]), W) + b)

# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])
```

success metrics

```
# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct,tf.float32))
```

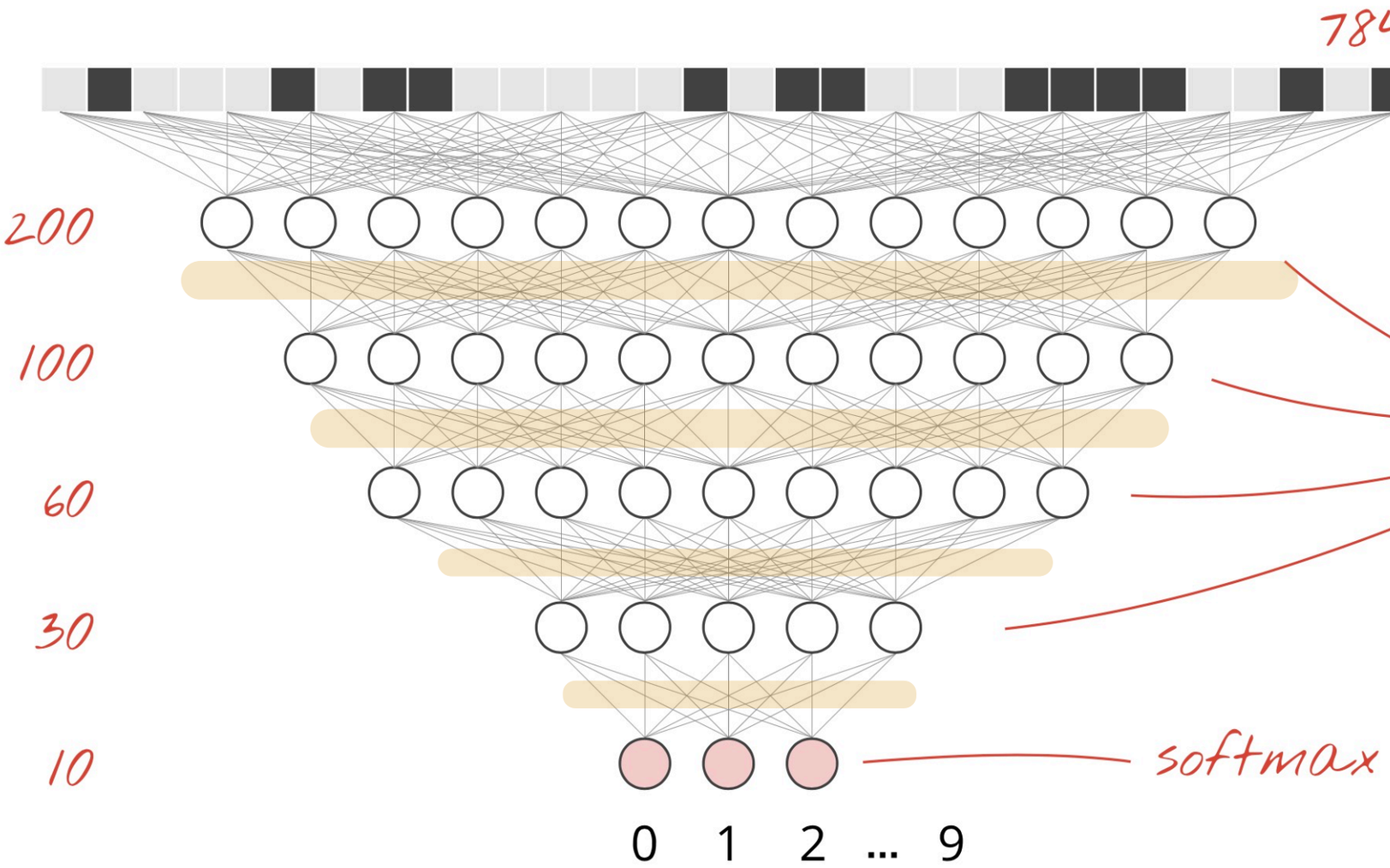
training step

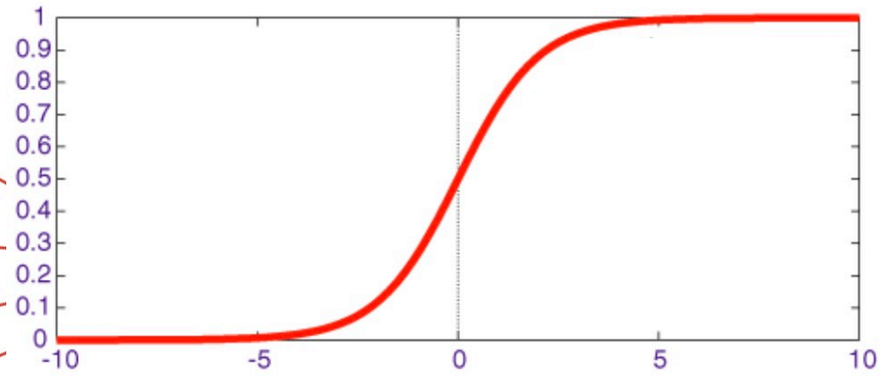
```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)
```

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

for i in range(10000):
      # load batch of images and correct answers
      batch_X, batch_Y = mnist.train.next_batch(100)
      train_data={X: batch_X, Y_: batch_Y}

      # train
      sess.run(train_step, feed_dict=train_data)

      # success ? add code to print it
      a,c = sess.run([accuracy, cross_entropy], feed=train_data)

      # success on test data ?
      test_data={X:mnist.test.images, Y_:mnist.test.labels}
      a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```

Run

Go deep! 5-layer NN





sigmoid function

$$F(x) = \frac{1}{1 + e^{-x}}$$

Note: 5 layers are overkill for this task

Init

```
K = 200
L = 100
M = 60
N = 30

W1 = tf.Variable(tf.truncated_normal([28*28, K] ,stddev=0.1))
B1 = tf.Variable(tf.zeros([K]))
W2 = tf.Variable(tf.truncated_normal([K, L], stddev=0.1))
B2 = tf.Variable(tf.zeros([L]))
W3 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B3 = tf.Variable(tf.zeros([M]))
W4 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B4 = tf.Variable(tf.zeros([N]))
W5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))
```

weights initialised with random values

Model

weights and biases

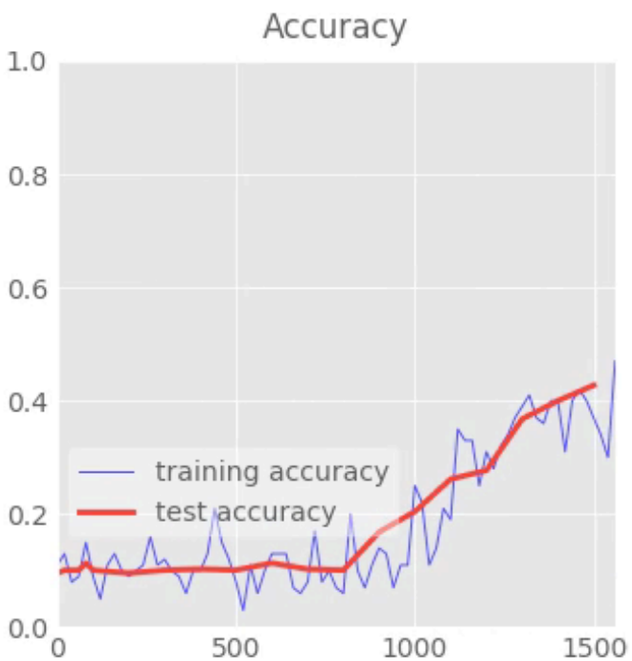
```
X = tf.reshape(X, [-1, 28*28])
```

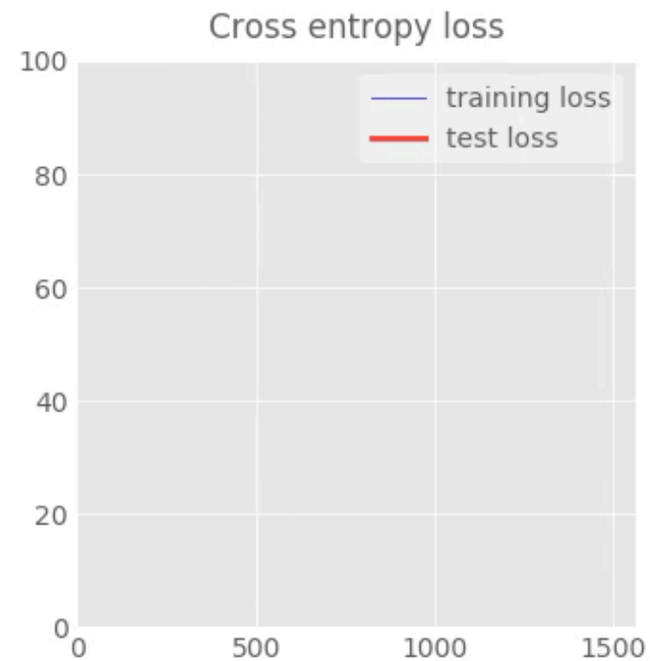
.. yet with a different activation function

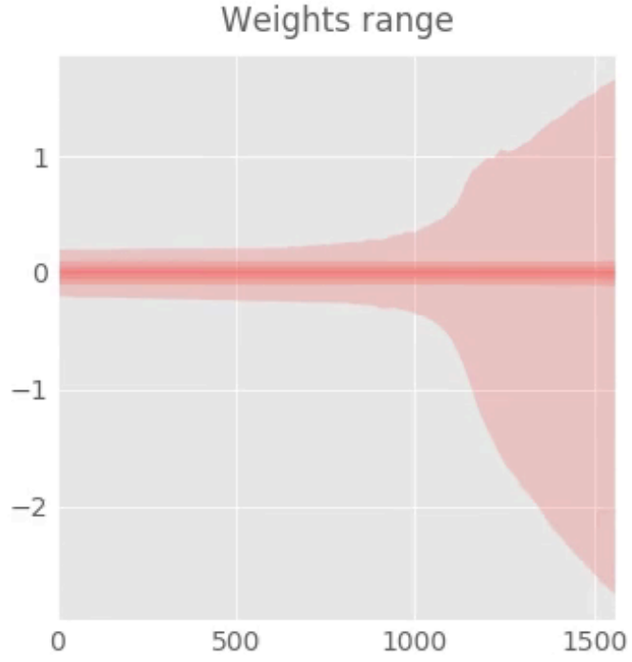
our previous unique layer

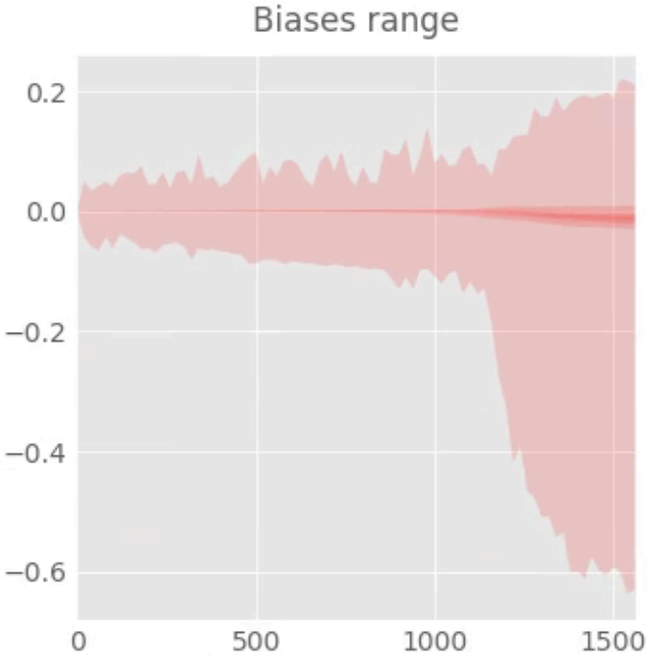
```
Y1 = tf.nn.sigmoid(tf.matmul(X, W1) + B1)
Y2 = tf.nn.sigmoid(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.sigmoid(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.sigmoid(tf.matmul(Y3, W4) + B4)
Y = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

Demo: mnist_2.0_five_layers_sigmoid.py

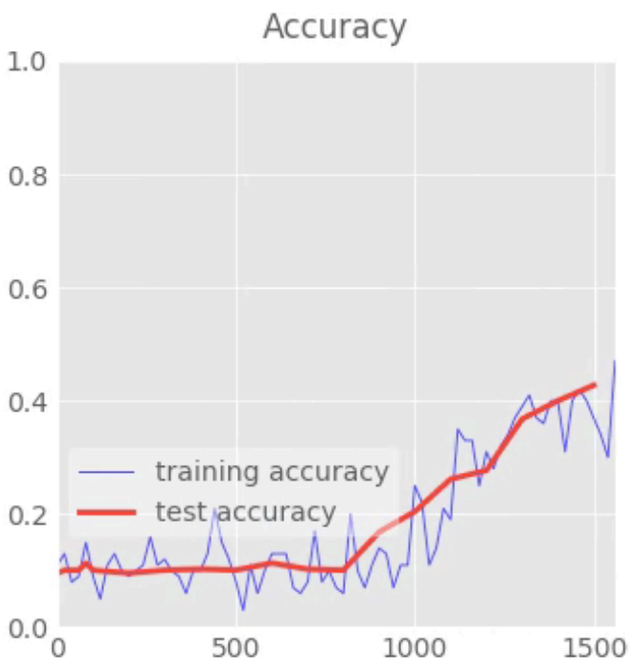


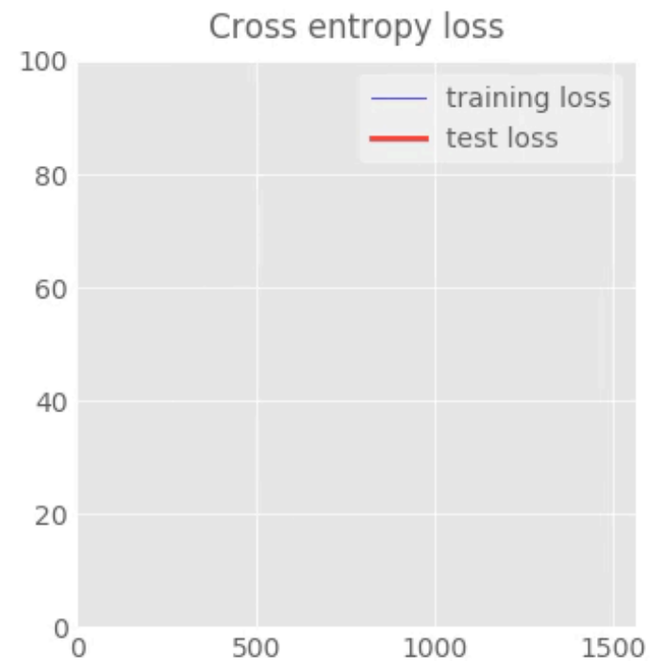


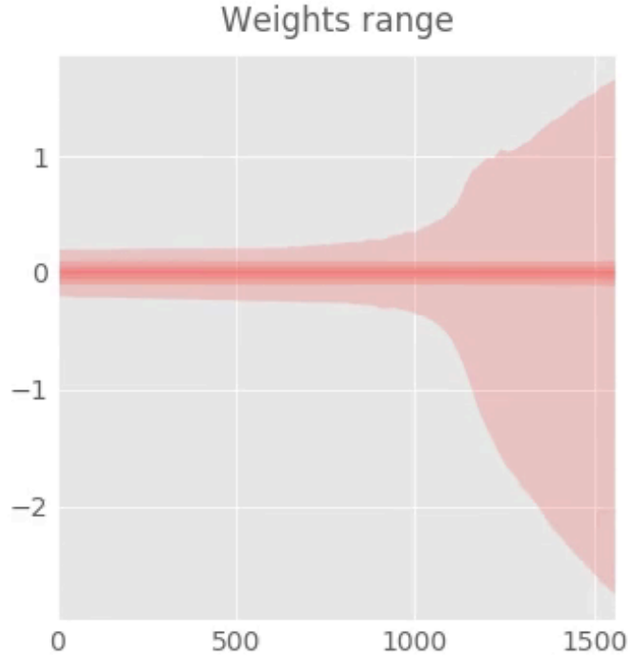


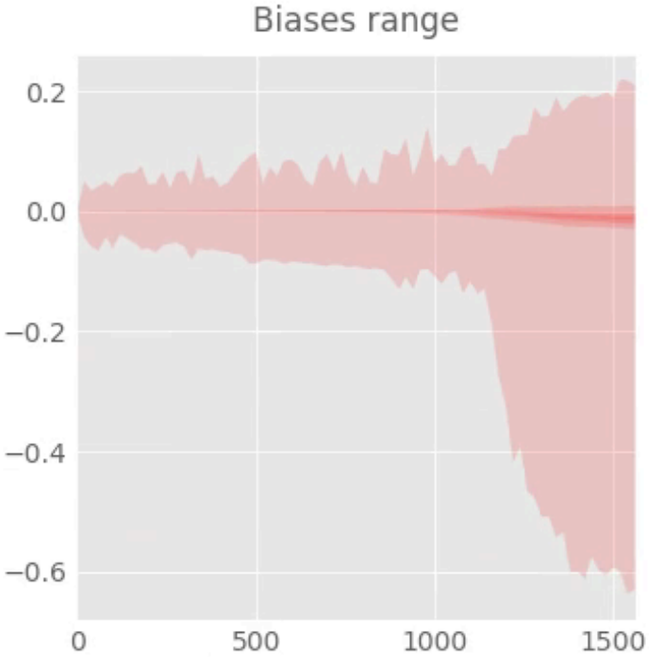


Demo: mnist_2.0_five_layers_sigmoid.py

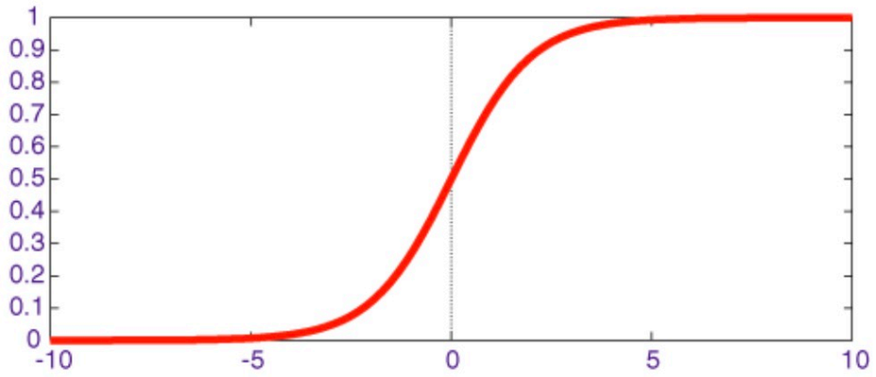








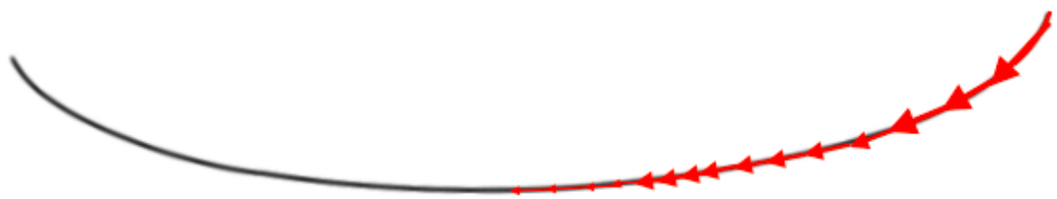
Slow convergence



sigmoid function

$$F(x) = \frac{1}{1 + e^{-x}}$$

descending a flat curve

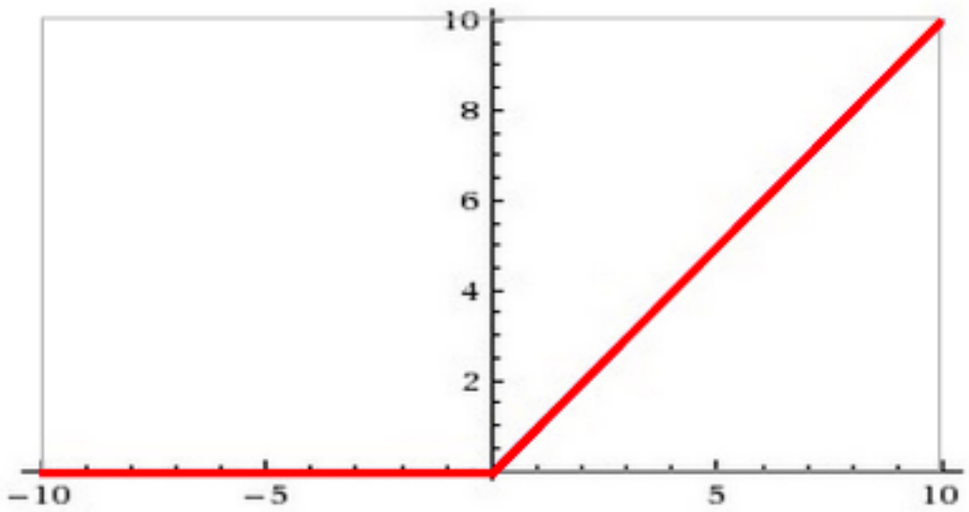


Vanishing gradients problem

1. If the gradient at each step is too small, then greater repetitions will be needed until convergence, because the weight is not changing enough at each iteration.
2. In **deep** networks, computing these gradients can involve taking the product of many small terms. and when you do so repeatedly, neuron outputs and their gradients can vanish entirely starting from the first layers (close to input).

The **ReLU** activation function can help prevent vanishing gradients.

ReLU



$$F(x) = \max(0, x)$$

How to use ReLU

Simply swap `tf.nn.sigmoid` with `tf.nn.relu` in code.

Biases initialization with small positive weights (just for ReLUs)

```
W = tf.Variable(tf.truncated_normal([K,L], stddev=0.1))
B = tf.Variable(tf.ones([L])/10)
```

A better optimizer

In very high dimensional spaces like here - we have in the order of 10K weights and biases - "saddle points" are frequent.

These are points that are not local minima but where the gradient is nevertheless zero and the gradient descent optimizer stays stuck there.

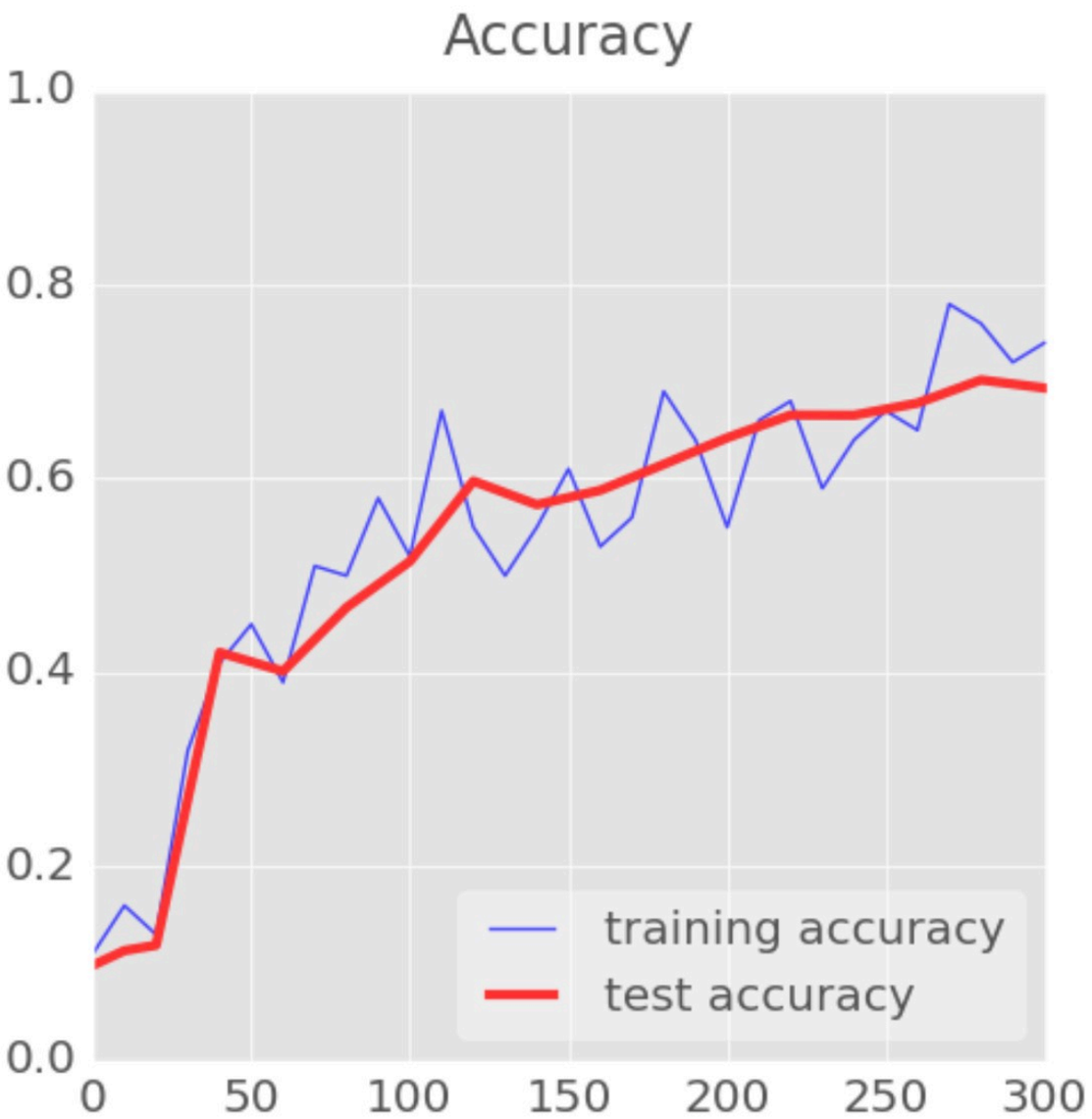
TensorFlow has a **full array** of available optimizers, including **some that work with an amount of inertia and will safely sail past saddle points**.

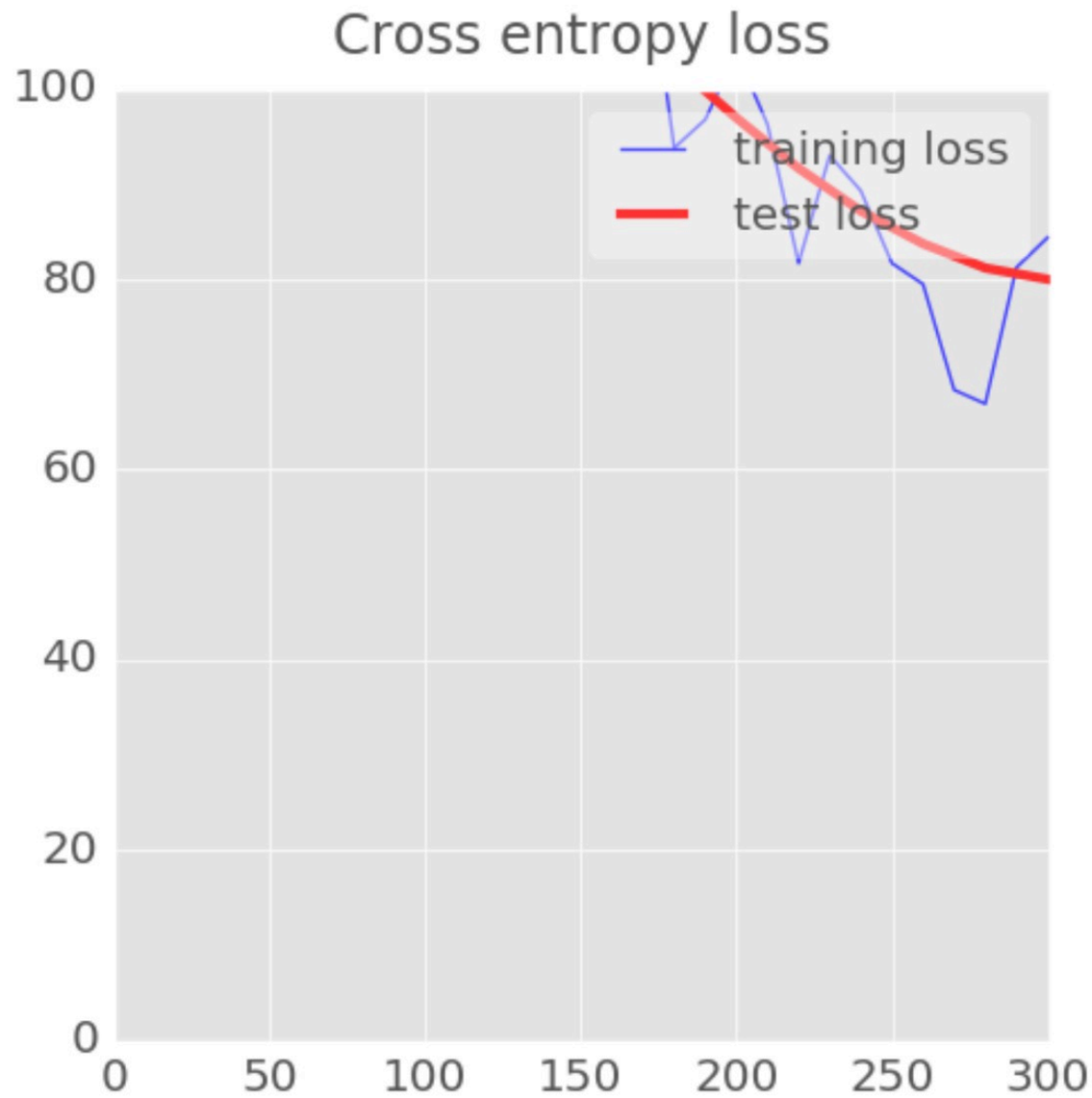
e.g.,

You can replace `tf.train.GradientDescentOptimiser` with a `tf.train.AdamOptimizer`

From sigmoid to ReLU

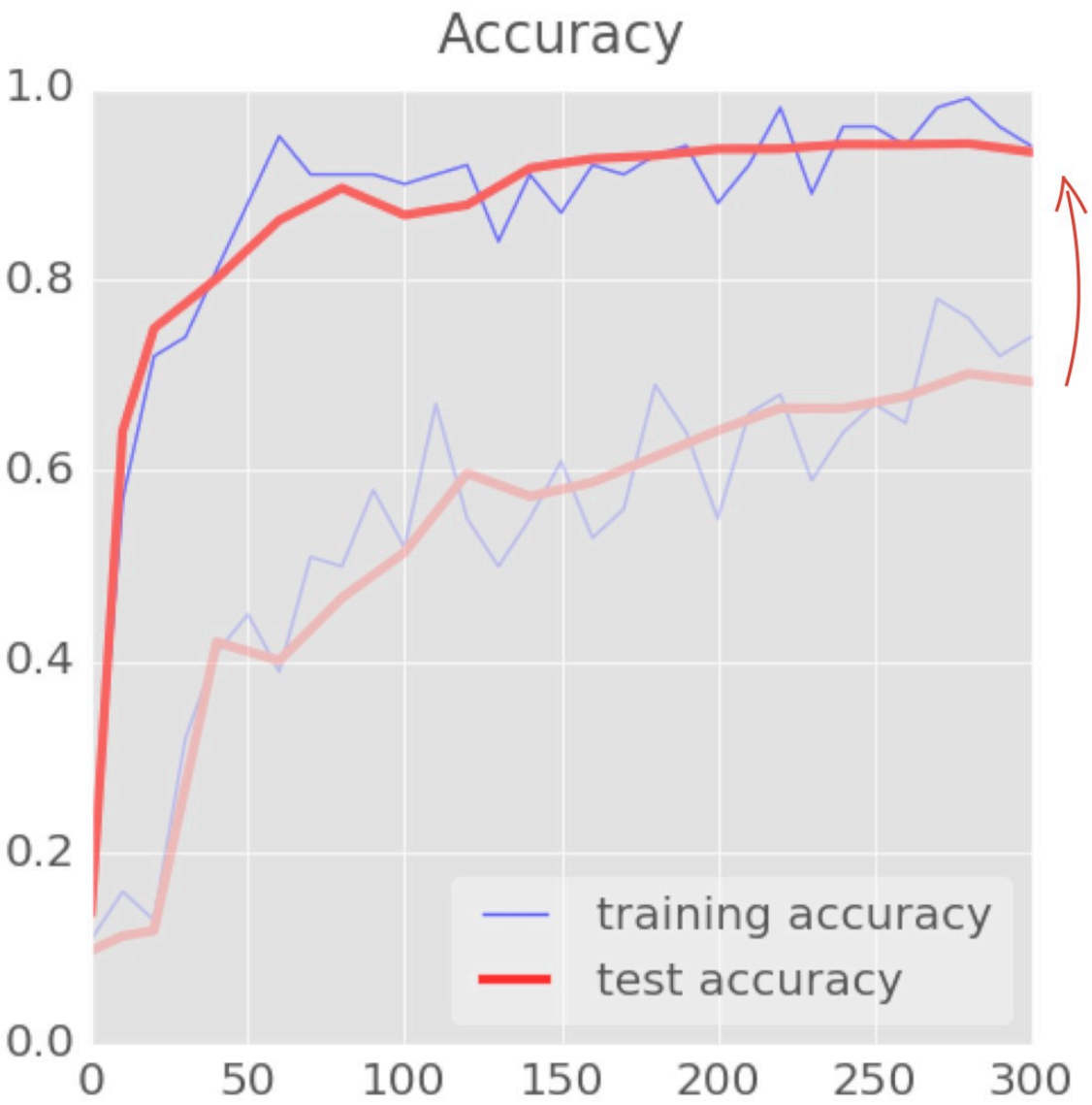
First 300 iterations

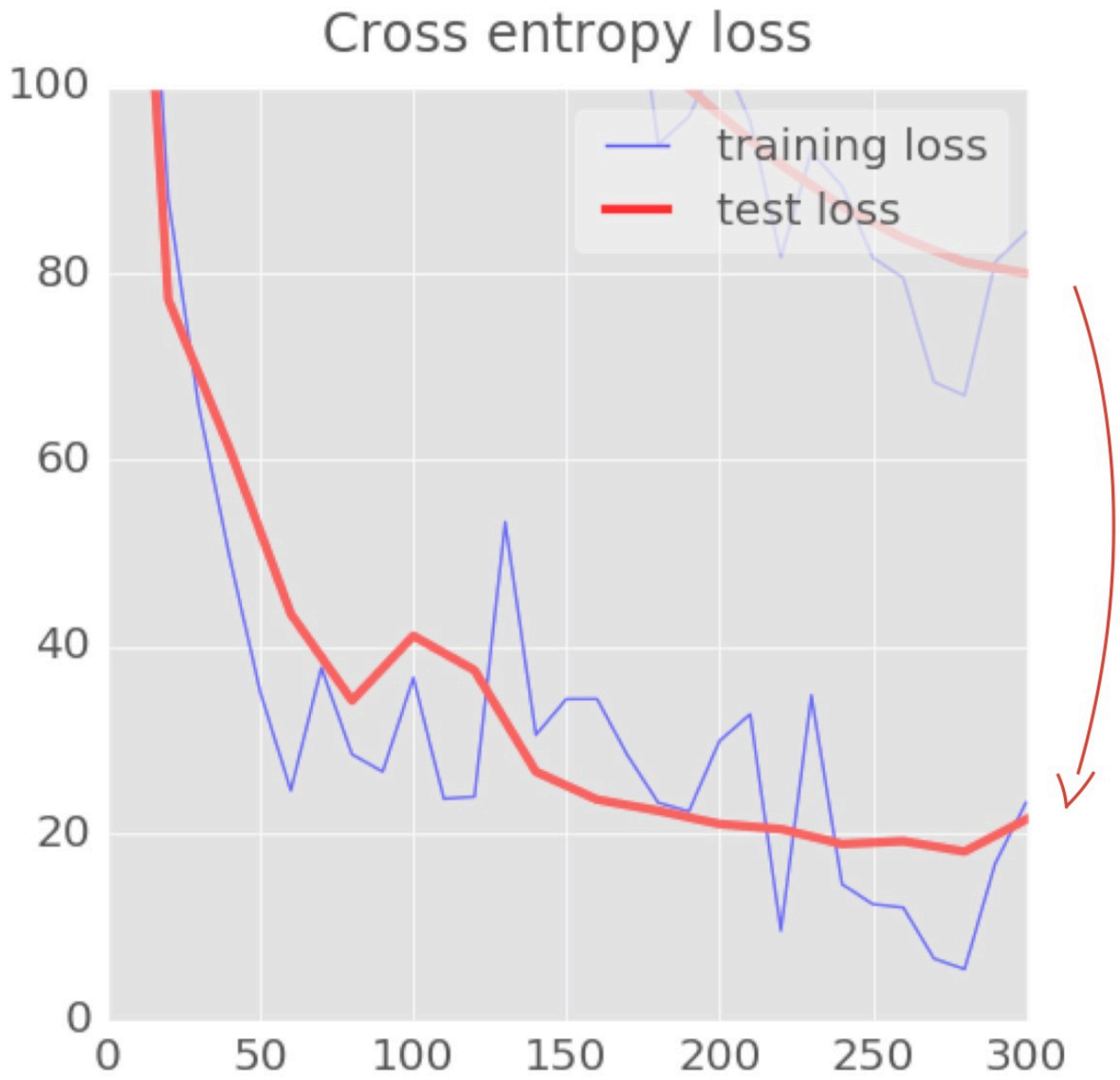




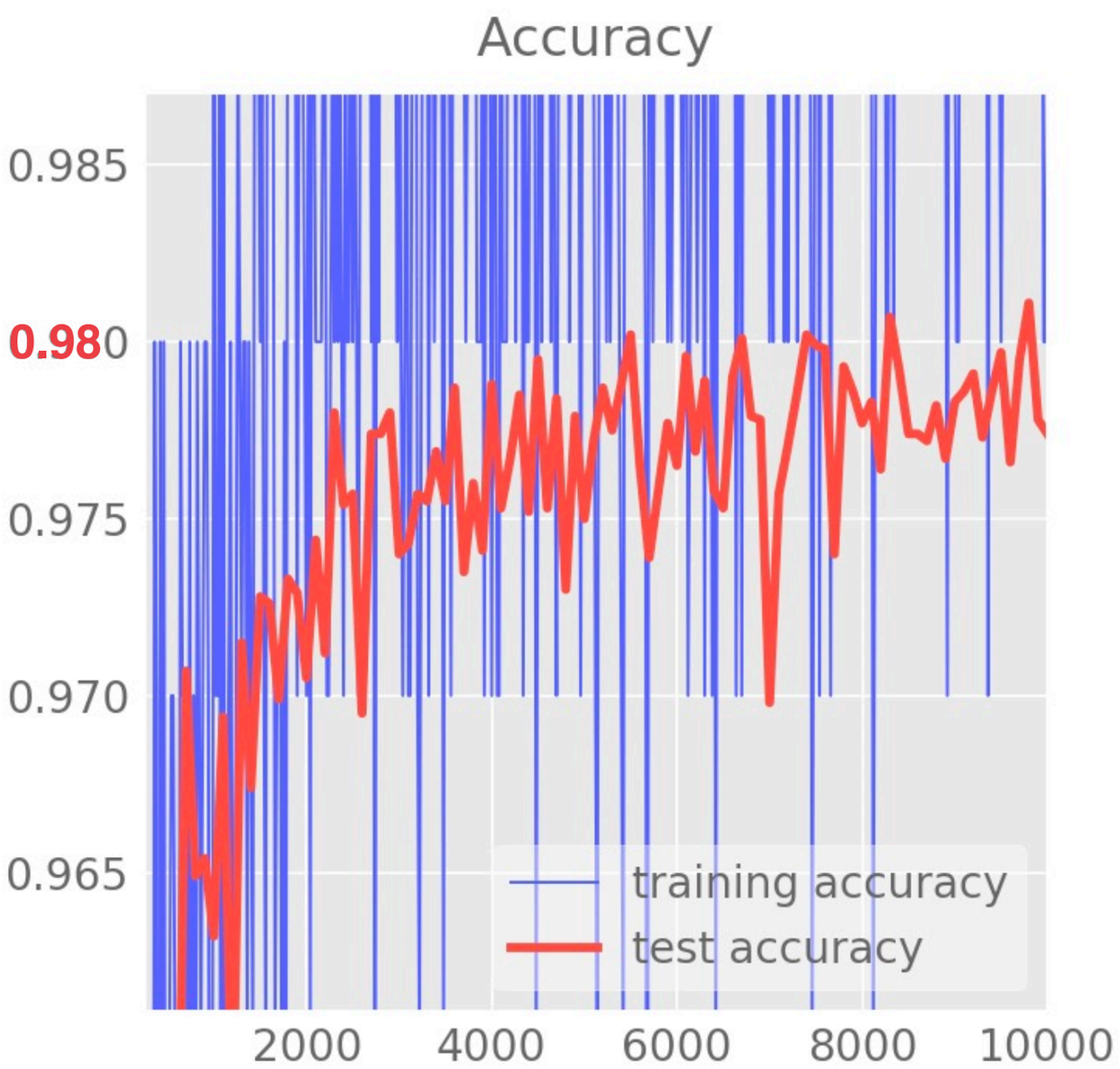
From sigmoid to ReLU

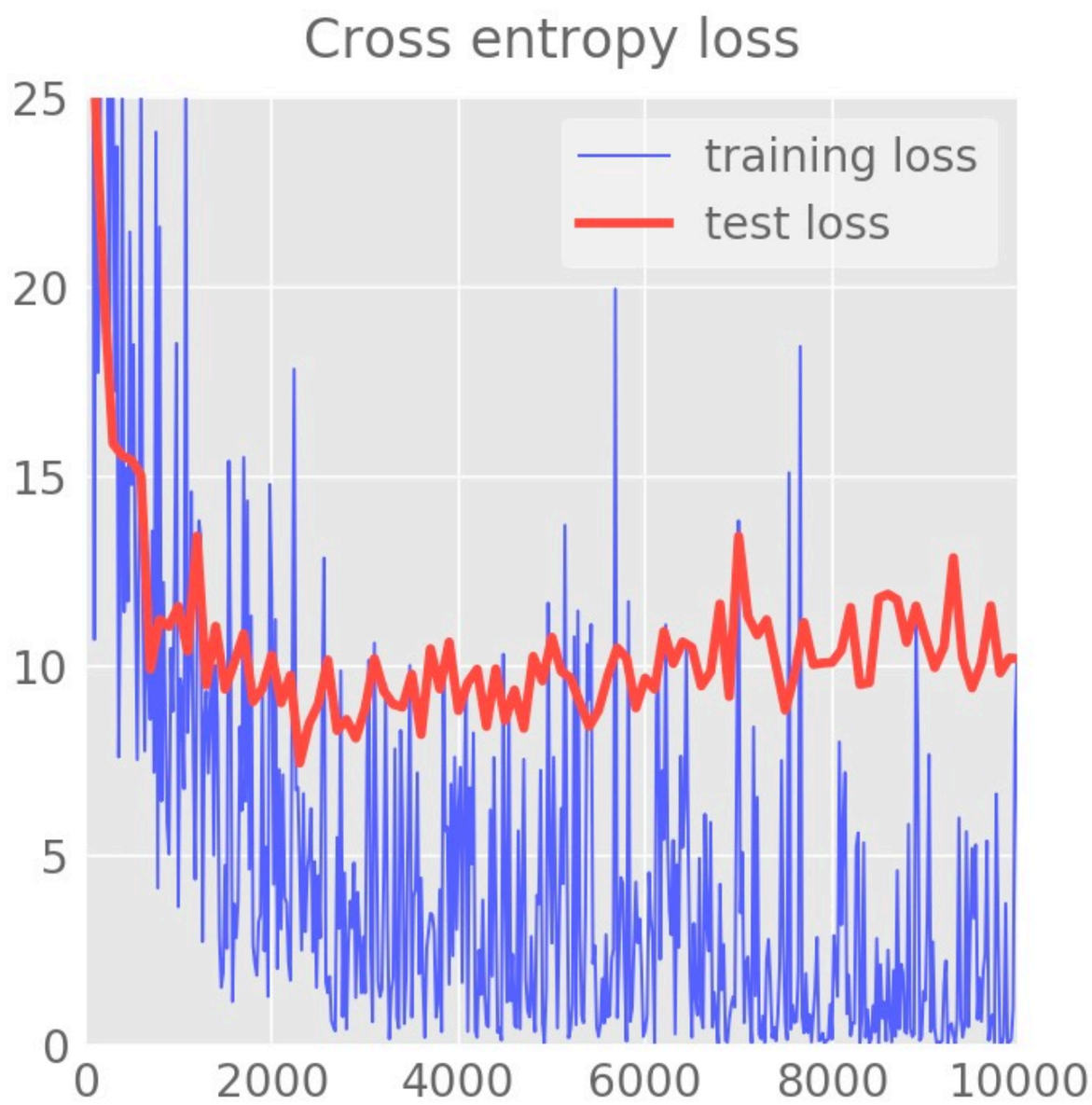
First 300 iterations





Noisy accuracy





Learning rate decay

If we reduce the learning rate, say by a factor of 10, we might slow down our learning by a factor of 10.

Right approach

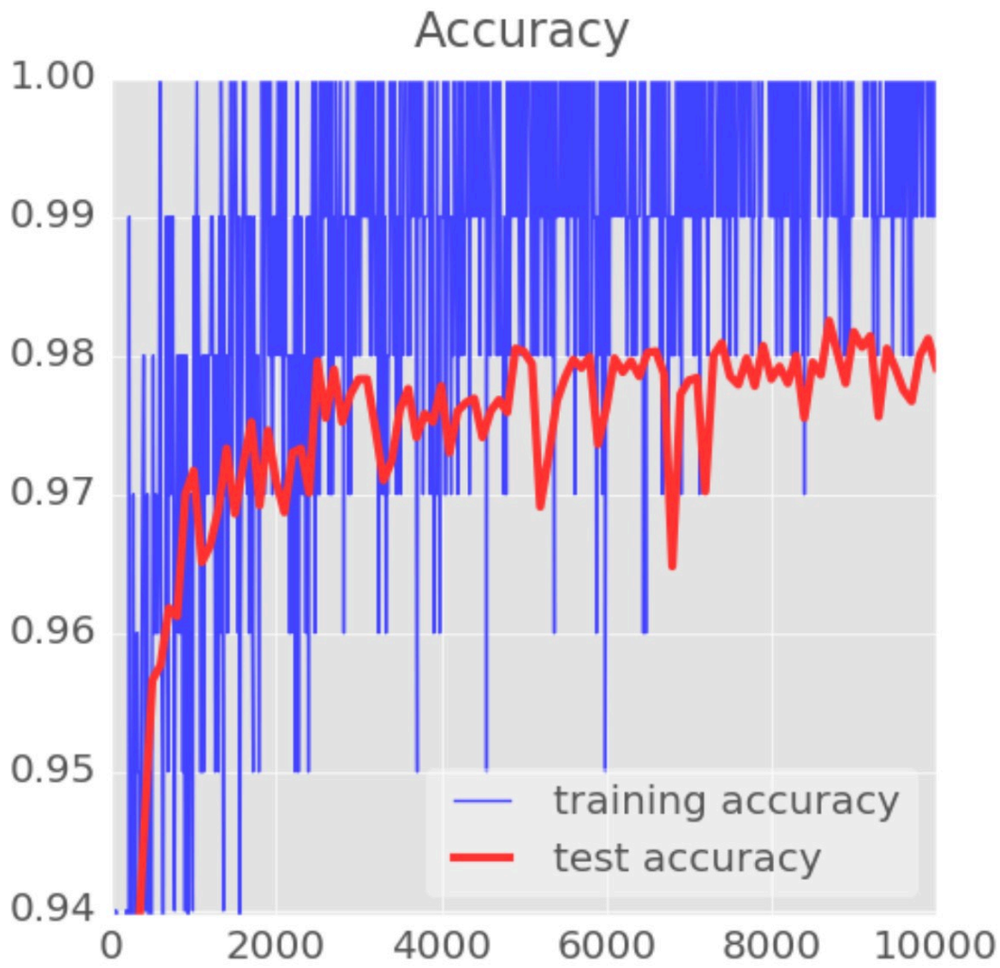
Start **fast**, then **decay**.

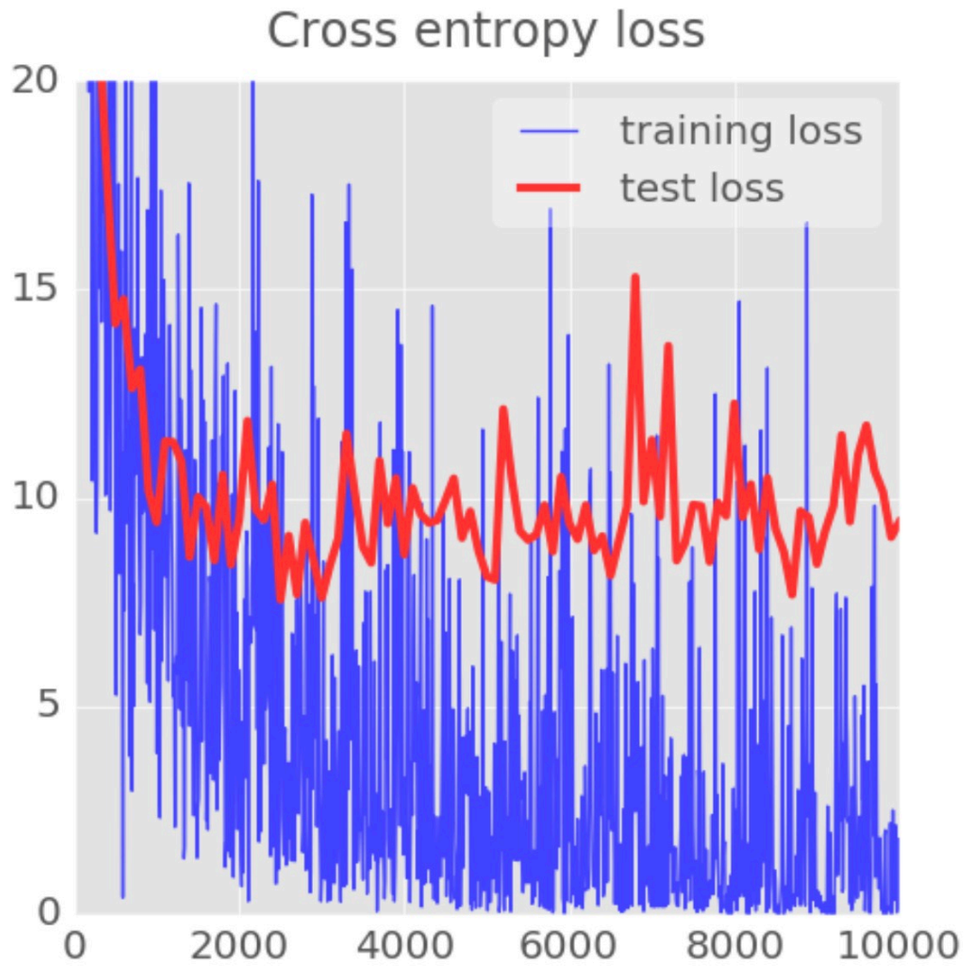
Learning rate decay

If we reduce the learning rate, say by a factor of 10, we might slow down our learning by a factor of 10.

Right approach

Start **fast**, then **decay**.





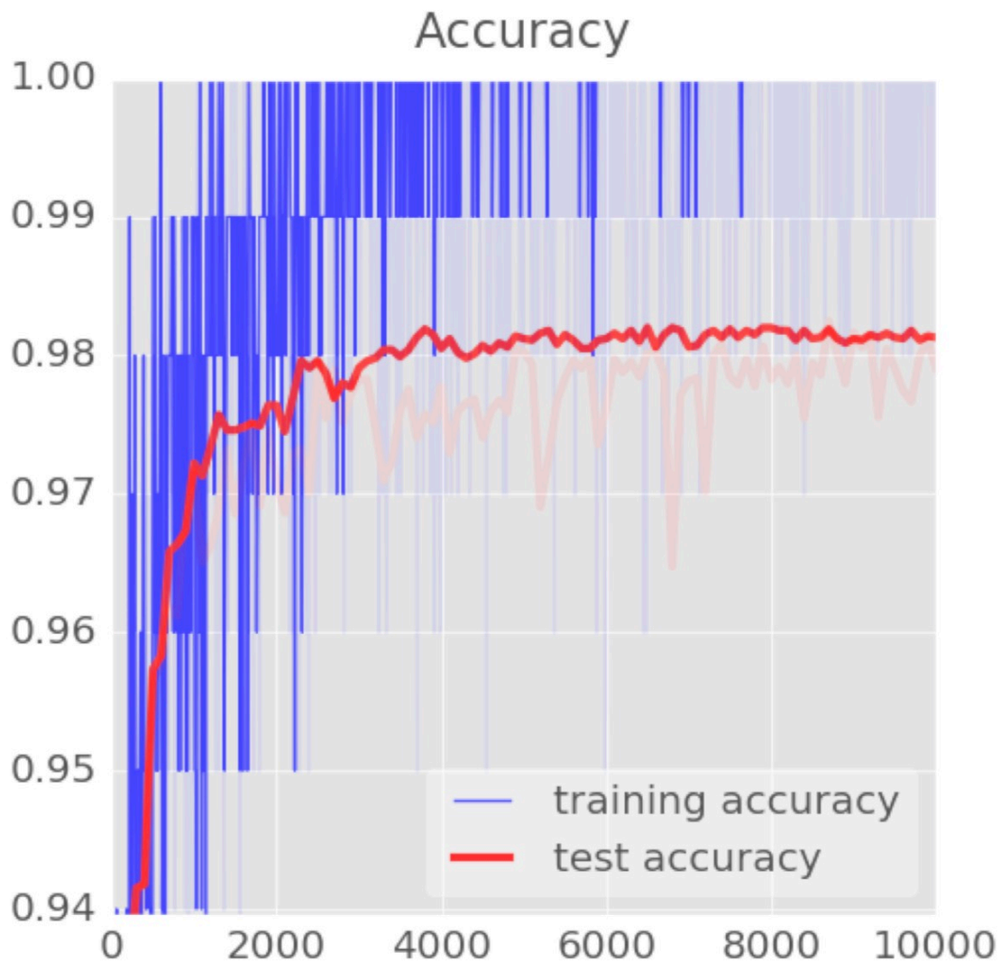
Fixed learning rate lr = 0.003 (no decay)

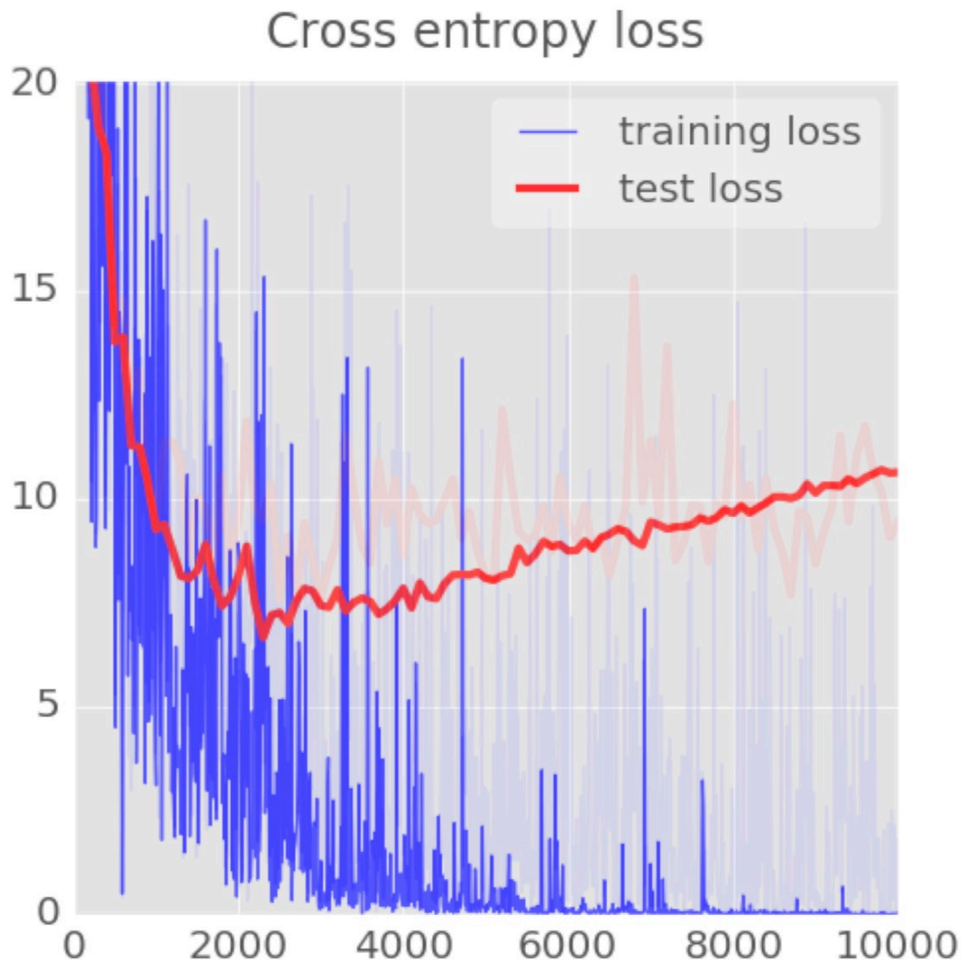
Learning rate decay

If we reduce the learning rate, say by a factor of 10, we might slow down our learning by a factor of 10.

Right approach

Start **fast**, then **decay**.

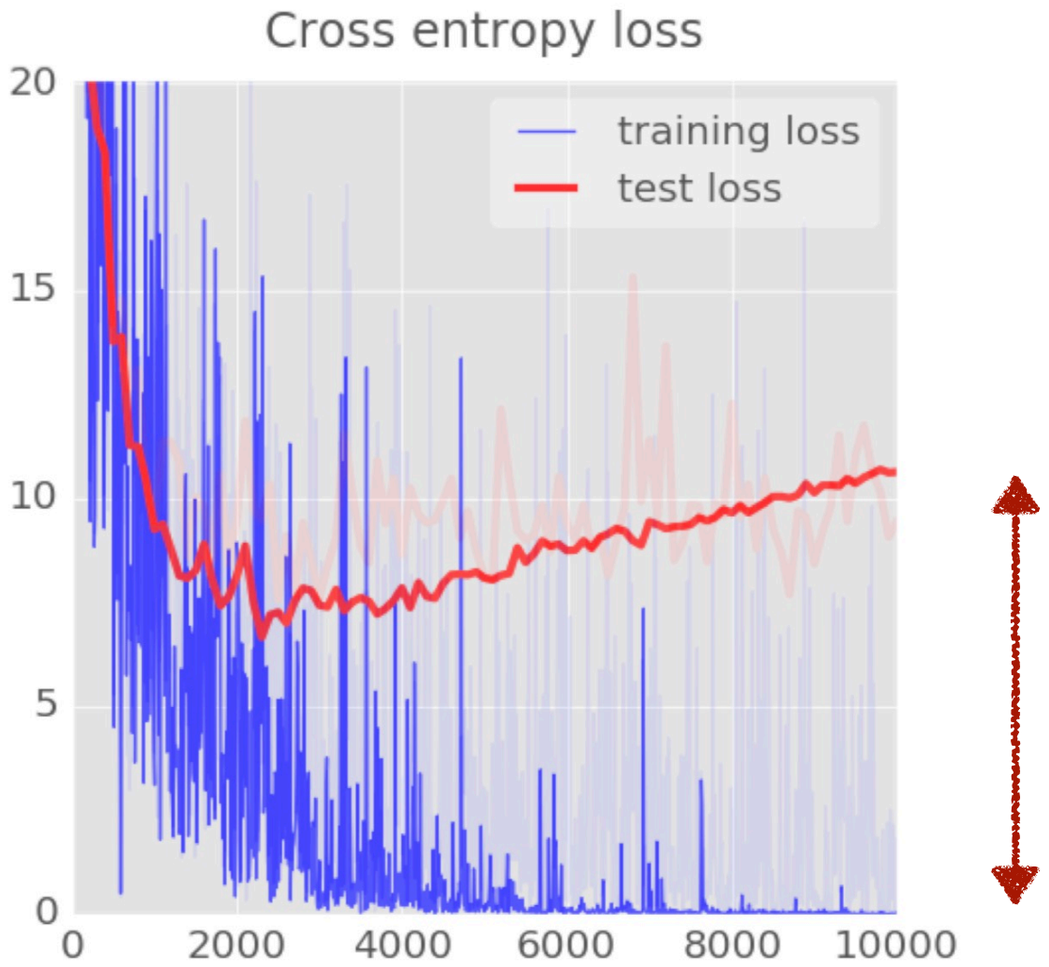




Start with lr=0.003 then drop exponentially to 0.0001

```
lr = lrmin+(lrmax-lrmin)*exp(-i/2000)
```

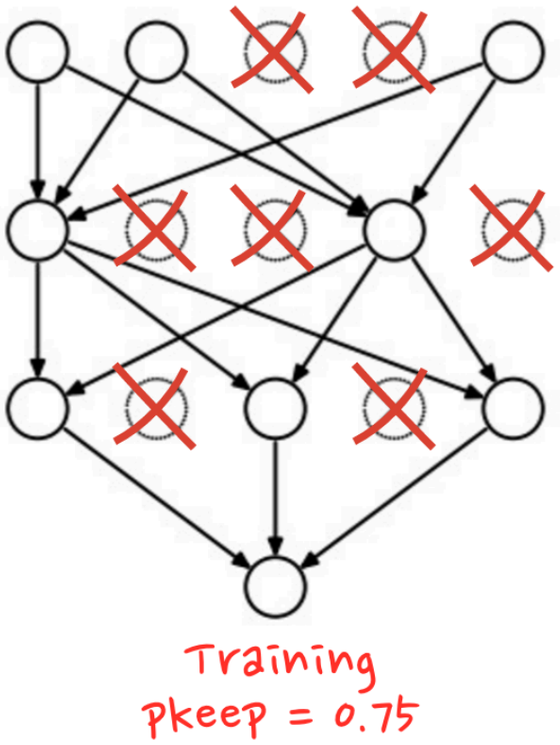
Overfitting?



The system is getting better and better on training data but performance on test data is not improving anymore

ML handbook solution**: regularization**

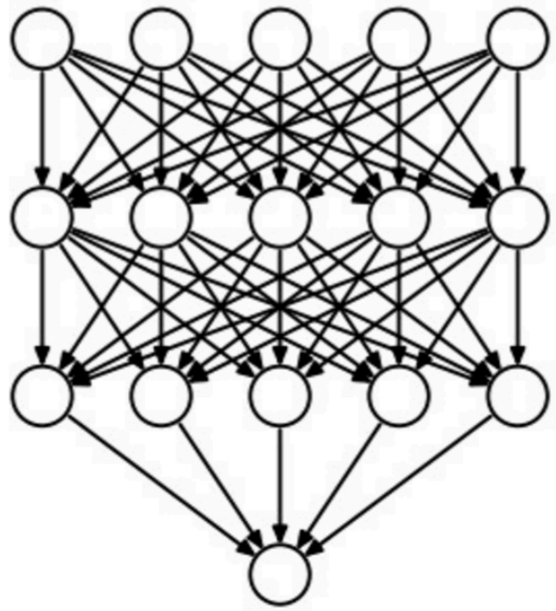
Dropout: shooting neurons



At each iteration:

drop fraction **(1-pkeep)** of neurons (with all their weights and biases)

(boosting the output of remaining neurons)



When testing, all neurons are put back in

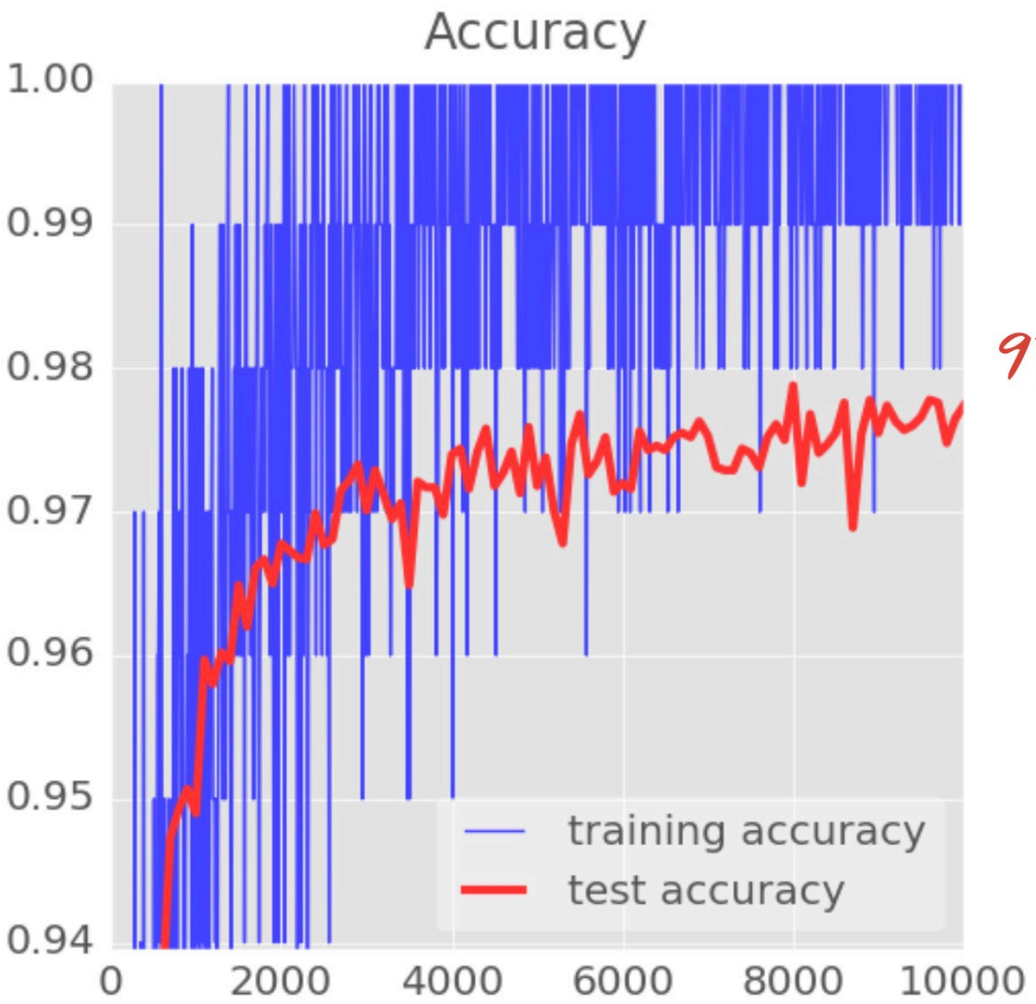
effect: those weights will not change for that iteration

TF code

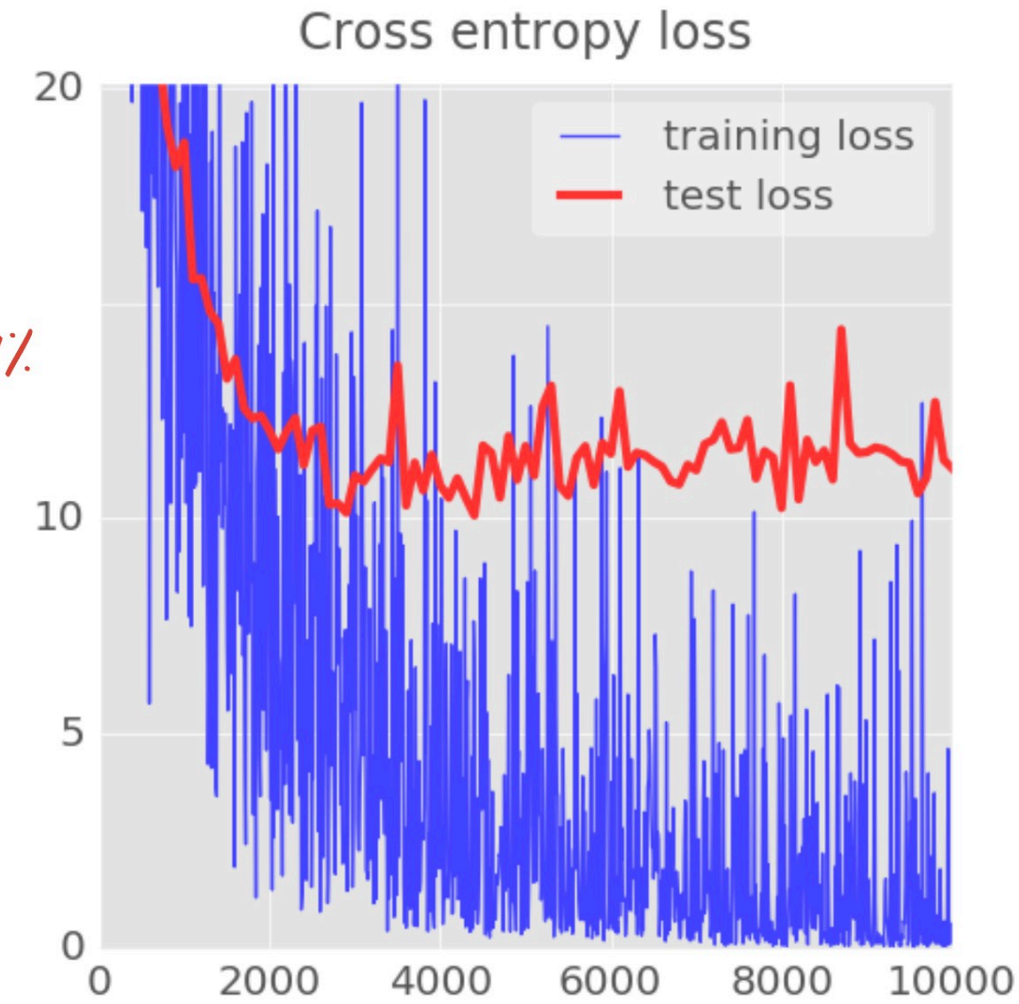
```
pkeep = tf.Placeholder(float32)
   Yf = tf.nn.relu(tf.matmul(X, W) + B)
   Y  = tf.nn.dropout(Yf, pkeep)
```

applied after each layer

All progress we have made so far

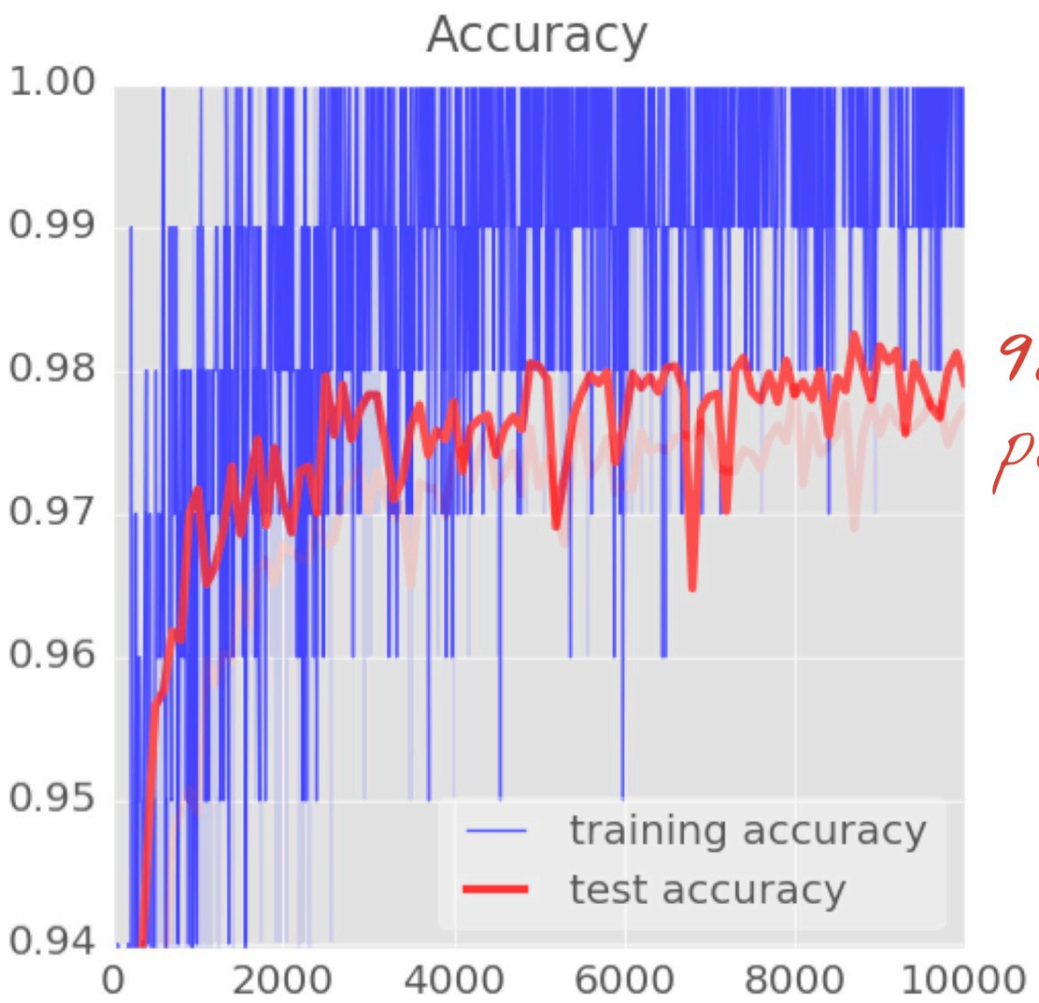


97.9%

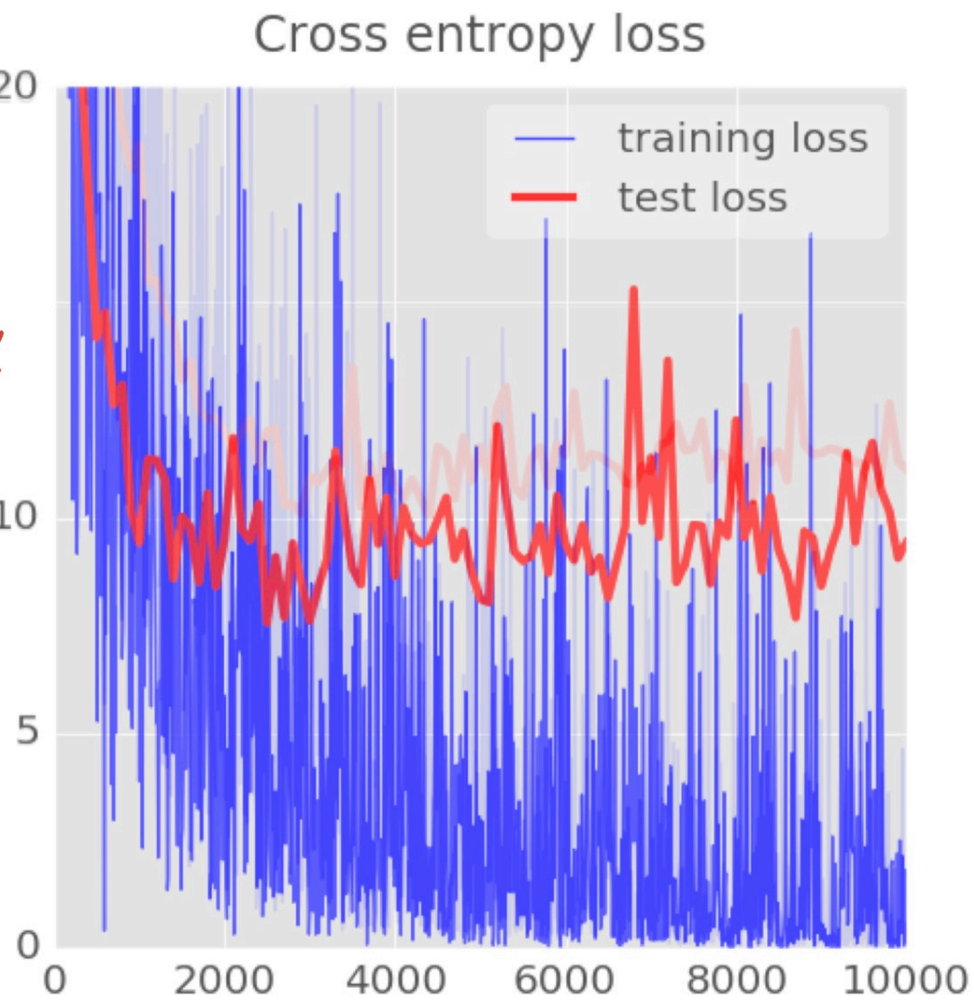


Sigmoid, learning rate = 0.003

All progress we have made so far

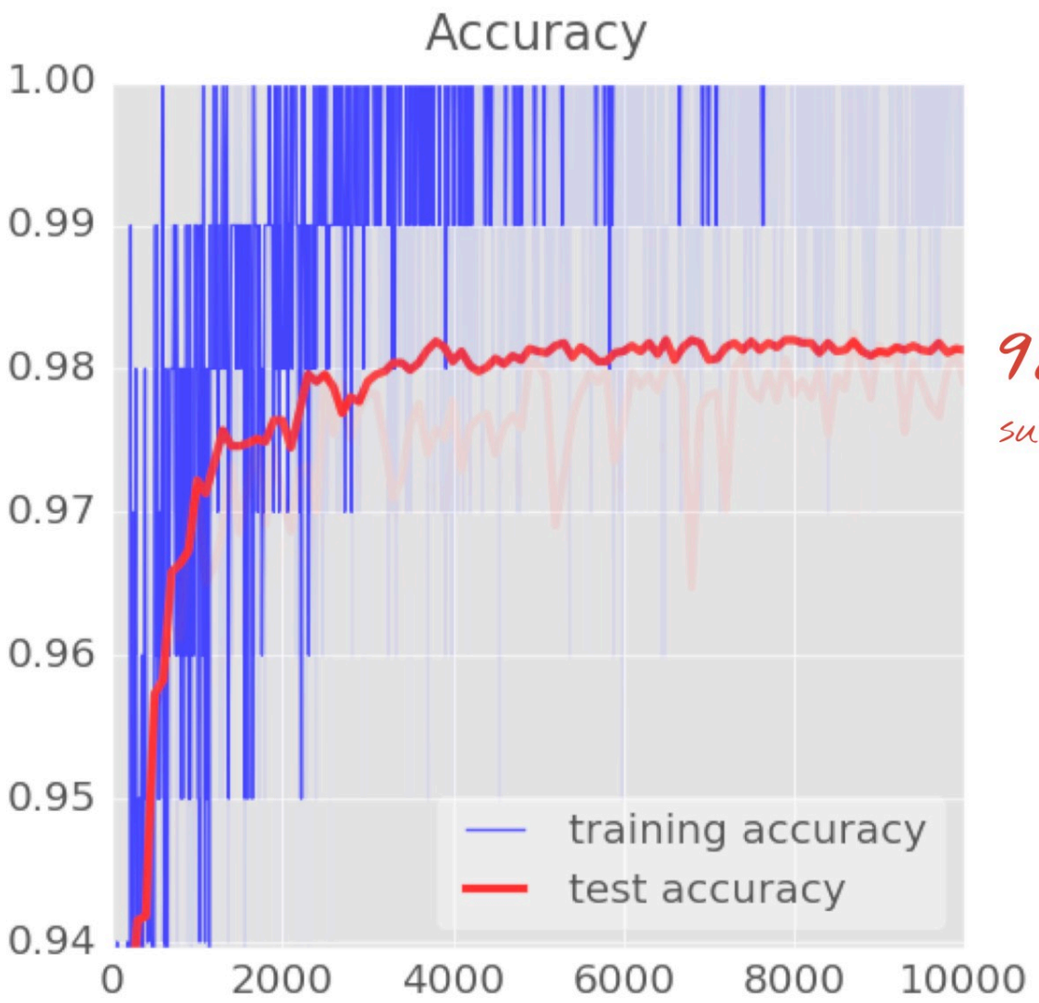


98.2% peak

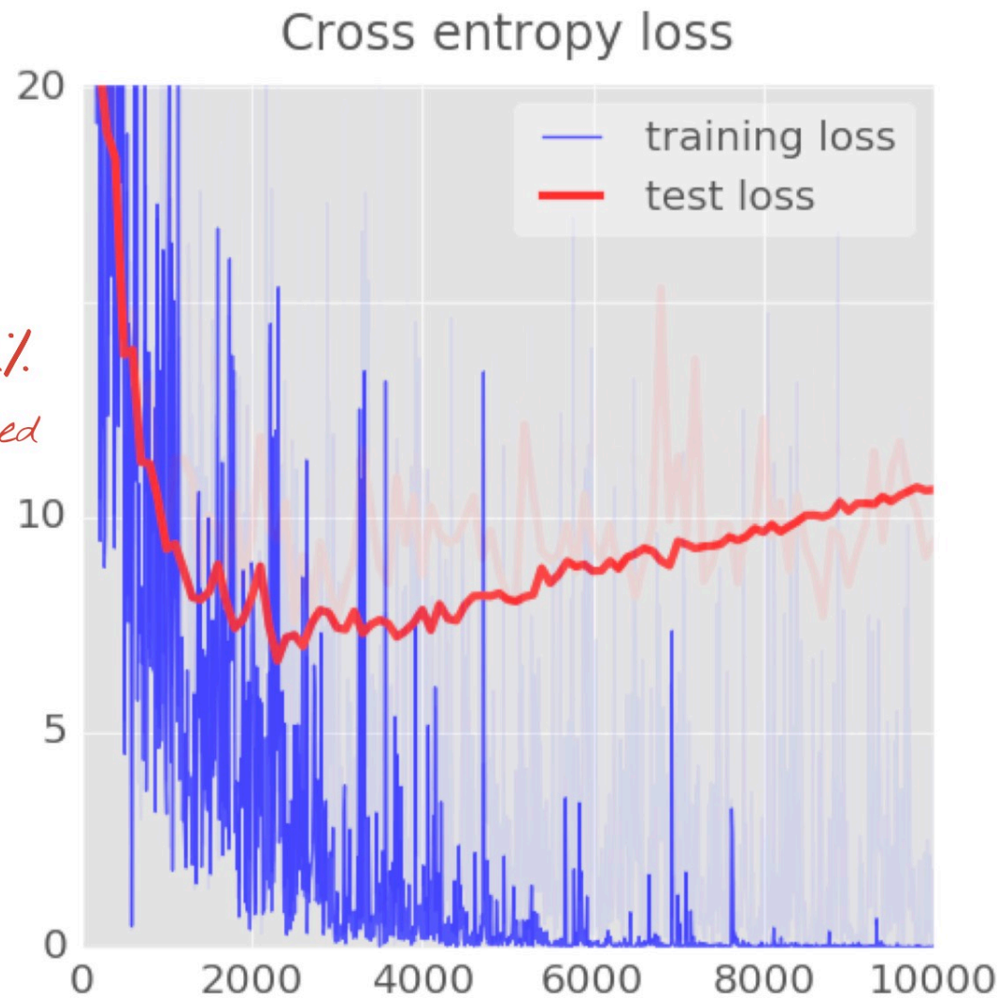


RELU, learning rate = 0.003

All progress we have made so far

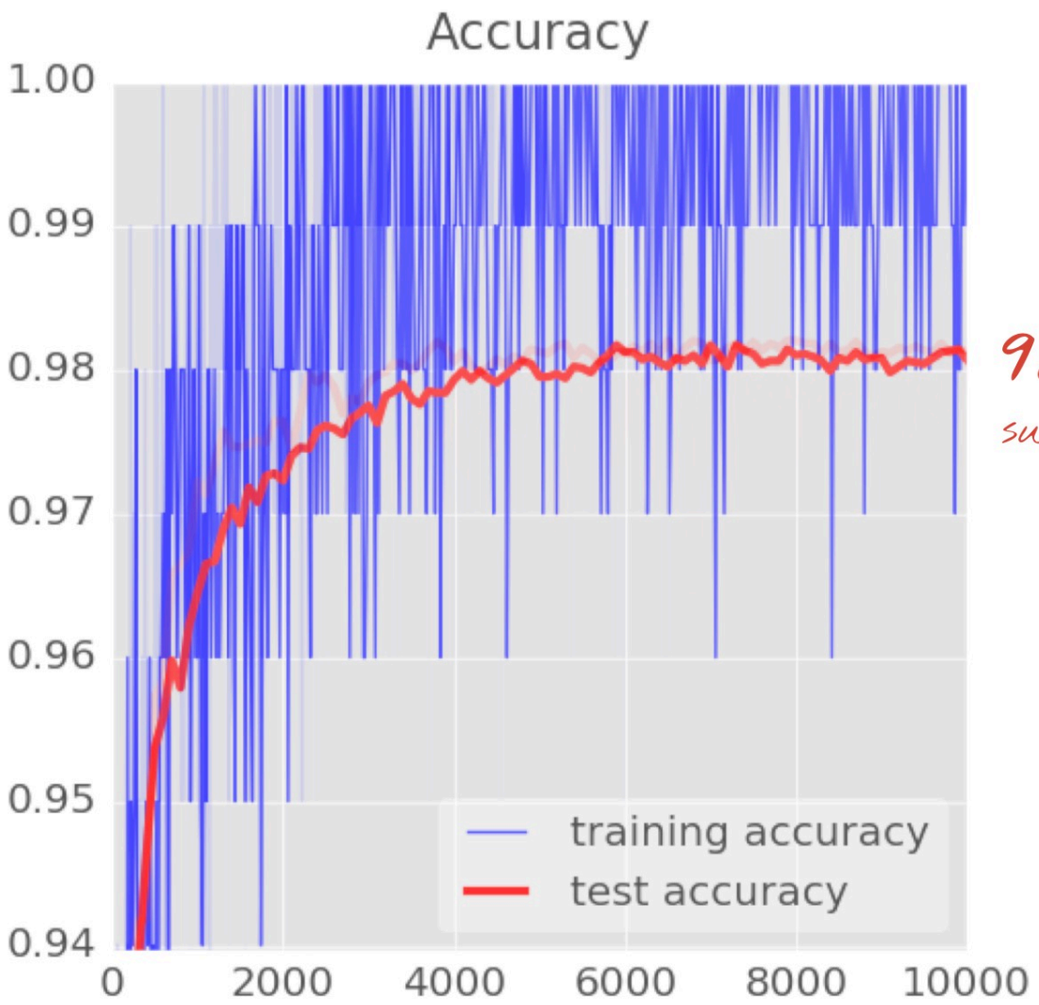


98.2% sustained



RELU, decaying learning rate 0.003 -> 0.0001

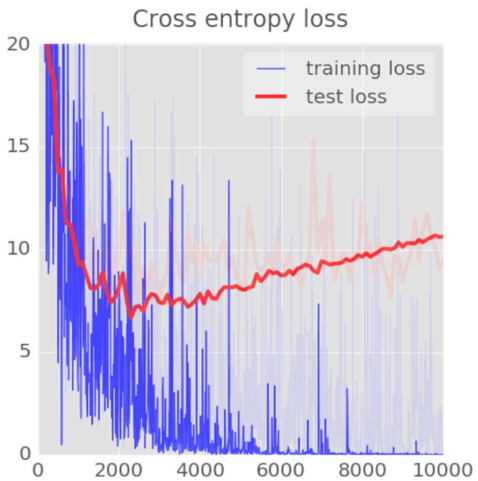
All progress we have made so far



98.2% sustained

RELU, decaying learning rate 0.003 -> 0.0001 and dropout 0.75

Causes of overfitting



Overfitting: the system is getting better and better on training data but performance on test data is not improving anymore.

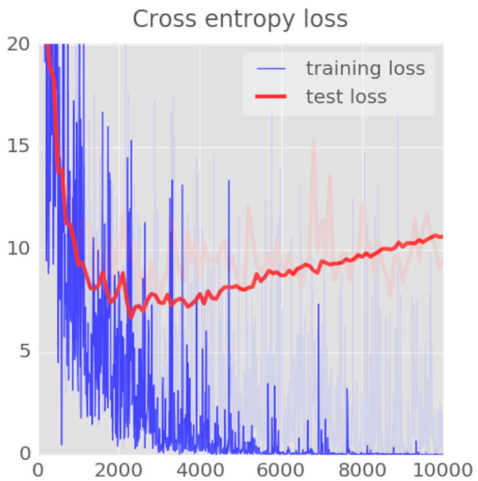
At its core overfitting happens when you have **too many degrees of freedom** (weights and biases) for the problem at hand.

Questions:

1. too few data? (no)
2. too many layers? (maybe)
3. **faulty design**?

At the present state, apparently by some faulty design our NN is not capable to constrain the d.o.f. and **extract the structure** we need. In other words it cannot learn *categories* in a sufficient way to generalize well to data it has never seen before.

Causes of overfitting



Overfitting: the system is getting better and better on training data but performance on test data is not improving anymore.

At its core overfitting happens when you have **too many degrees of freedom** (weights and biases) for the problem at hand.

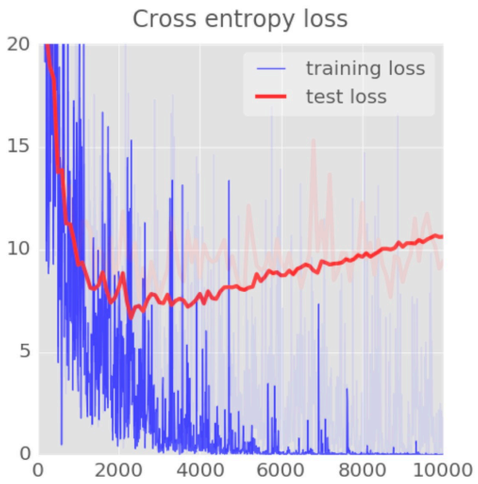
Questions:

1. too few data? (no)
2. too many layers? (maybe)
3. **faulty design**?

At the present state, apparently by some faulty design our NN is not capable to constrain the d.o.f. and **extract the structure** we need. In other words it cannot learn *categories* in a sufficient way to generalize well to data it has never seen before.

Why?

Causes of overfitting



Overfitting: the system is getting better and better on training data but performance on test data is not improving anymore.

At its core overfitting happens when you have **too many degrees of freedom** (weights and biases) for the problem at hand.

Questions:

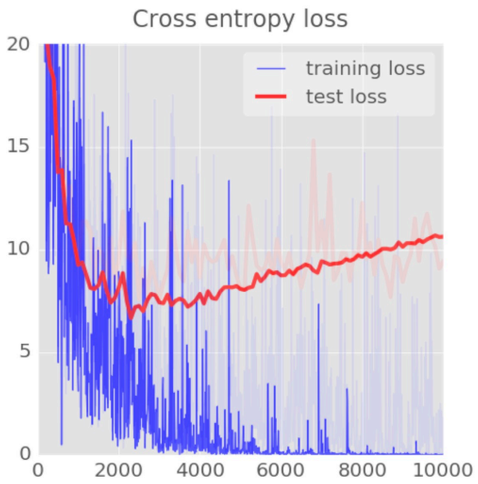
1. too few data? (no)
2. too many layers? (maybe)
3. **faulty design**?

At the present state, apparently by some faulty design our NN is not capable to constrain the d.o.f. and **extract the structure** we need. In other words it cannot learn *categories* in a sufficient way to generalize well to data it has never seen before.

Why?

Hint: We did **something very stupid**

Causes of overfitting



Overfitting: the system is getting better and better on training data but performance on test data is not improving anymore.

At its core overfitting happens when you have **too many degrees of freedom** (weights and biases) for the problem at hand.

Questions:

1. too few data? (no)
2. too many layers? (maybe)
3. **faulty design**?

At the present state, apparently by some faulty design our NN is not capable to constrain the d.o.f. and **extract the structure** we need. In other words it cannot learn *categories* in a sufficient way to generalize well to data it has never seen before.

Why?

Hint: We did **something very stupid** with the input.

When shape matters

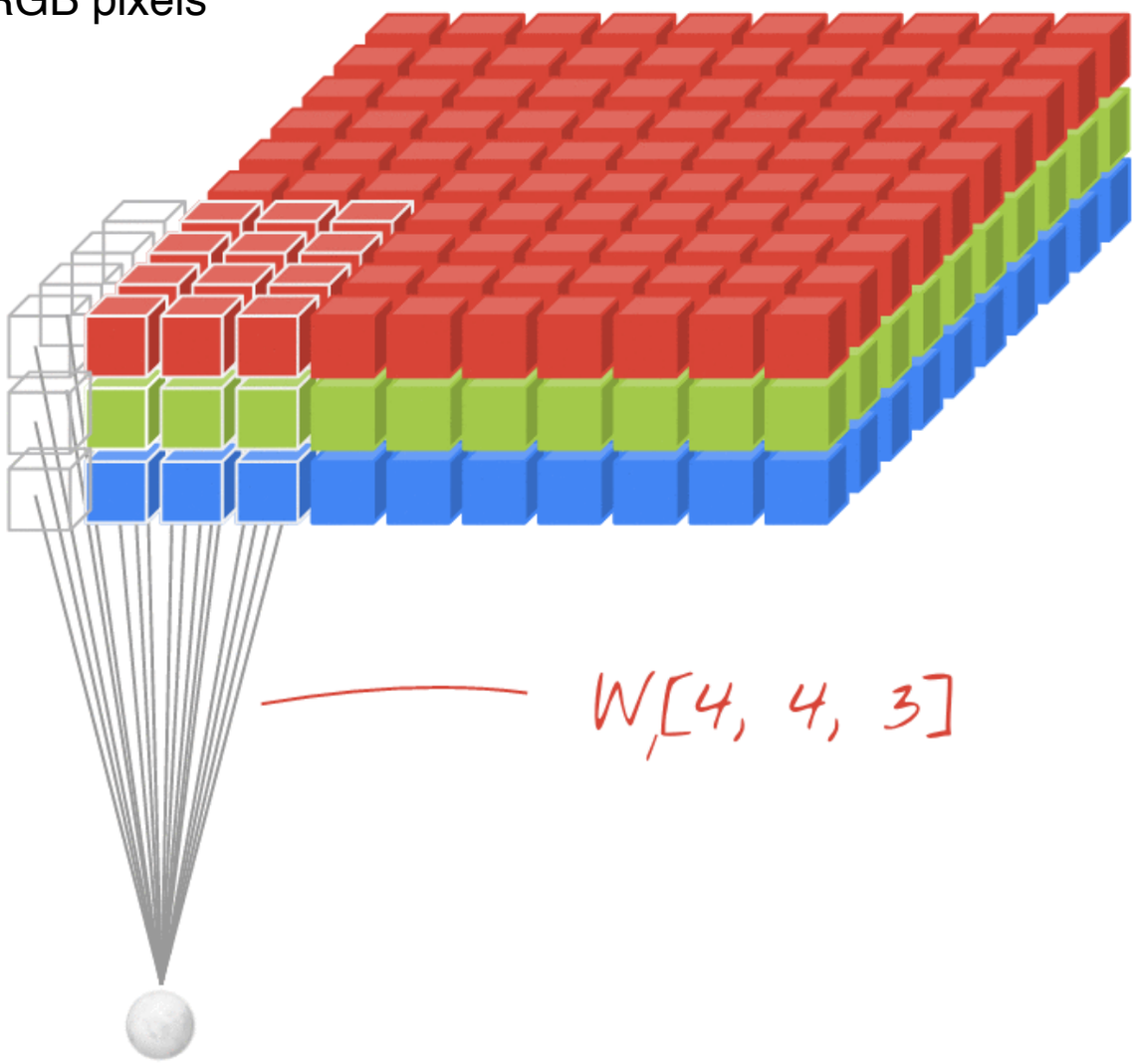
Remember how we are using our images, all pixels flattened into a single long vector ?
That was **a really bad idea**.

Handwritten digits are made of **shapes** and we discarded the shape information when we flattened the pixels.
However, there is a type of neural network that can take advantage of shape information: **convolutional networks**.

These network are specifically designed for > 1D datasets (images) where shape information (locality) is important.

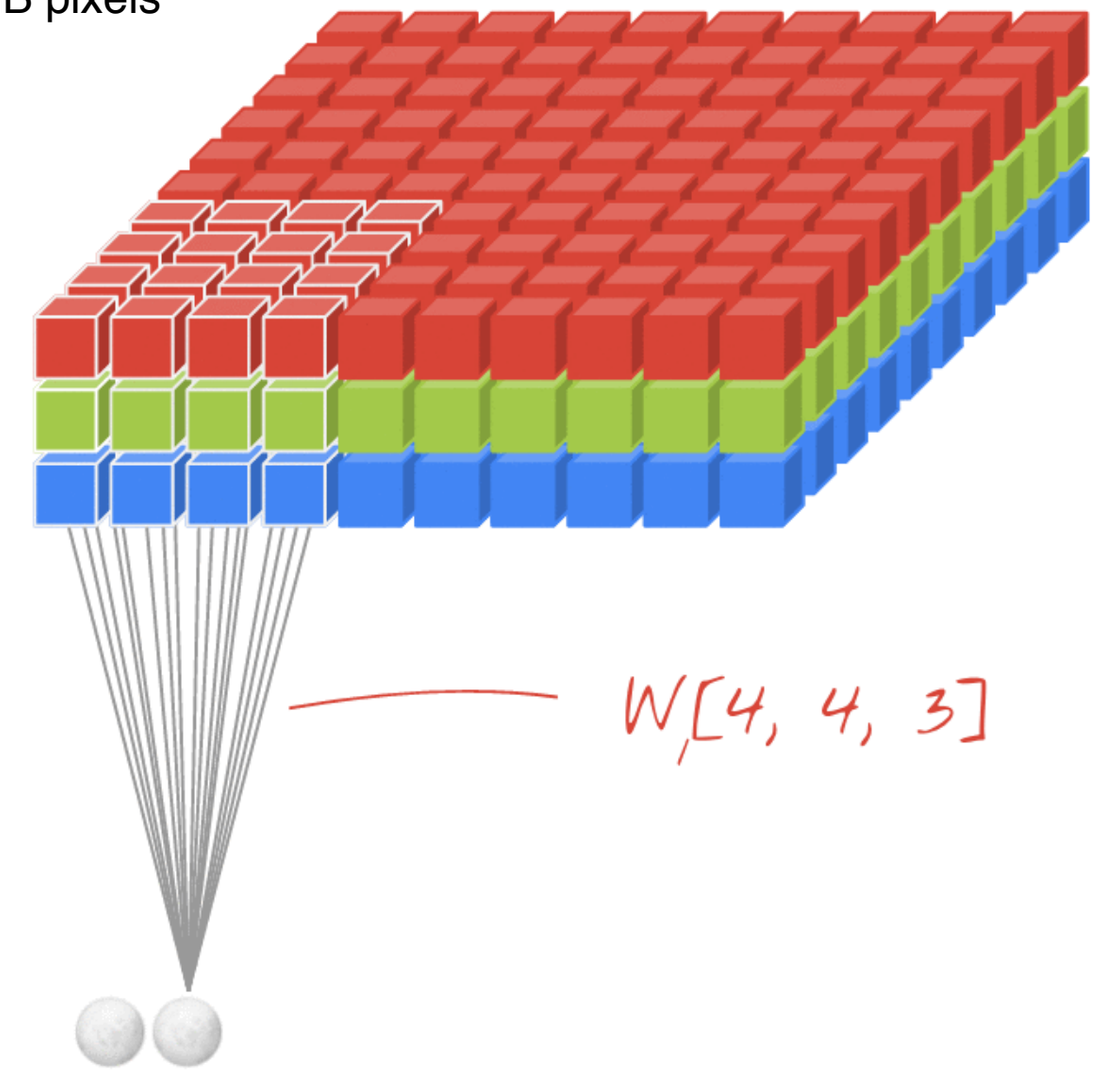
Getting up 98% accuracy: CNN

RGB pixels



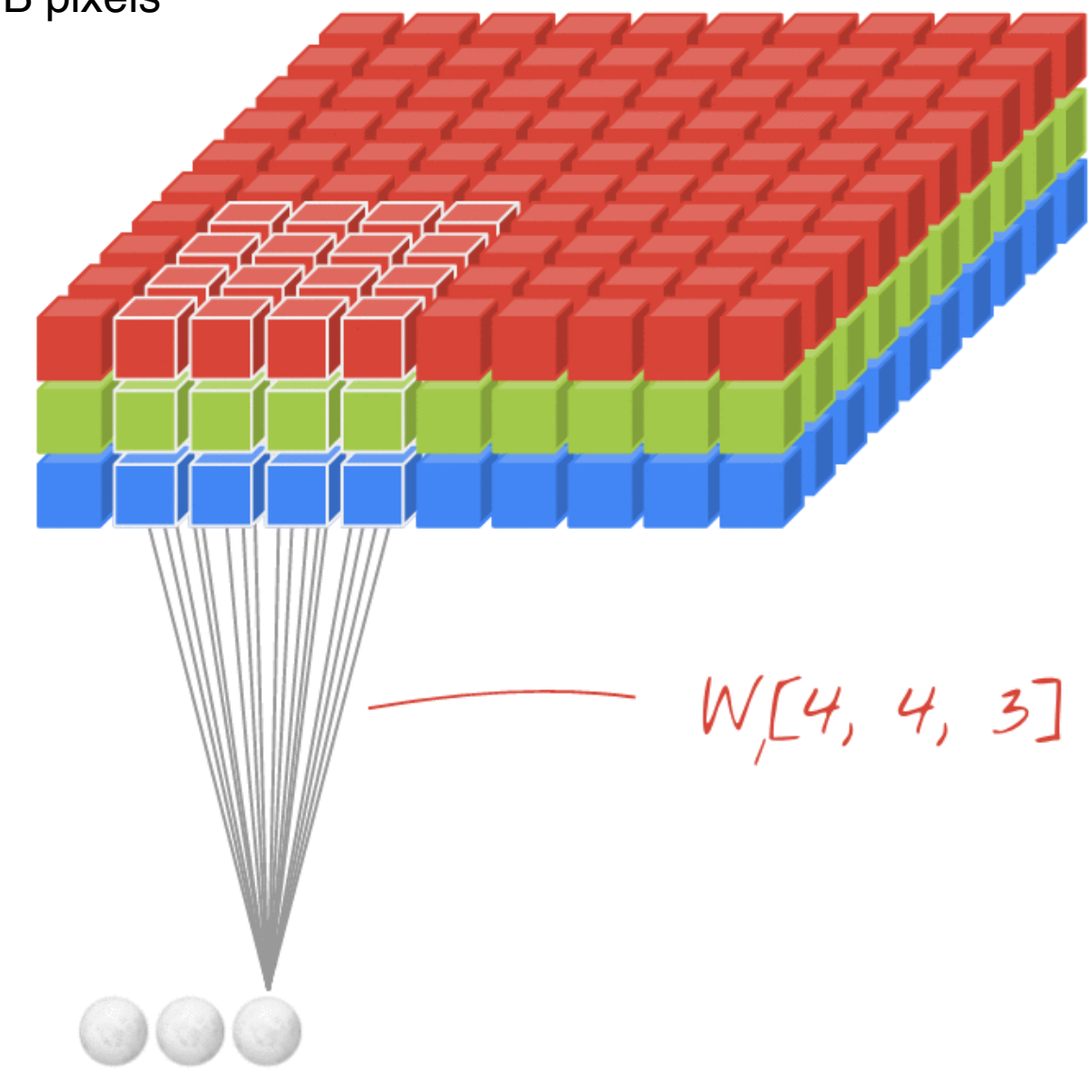
Getting up 98% accuracy: CNN

RGB pixels



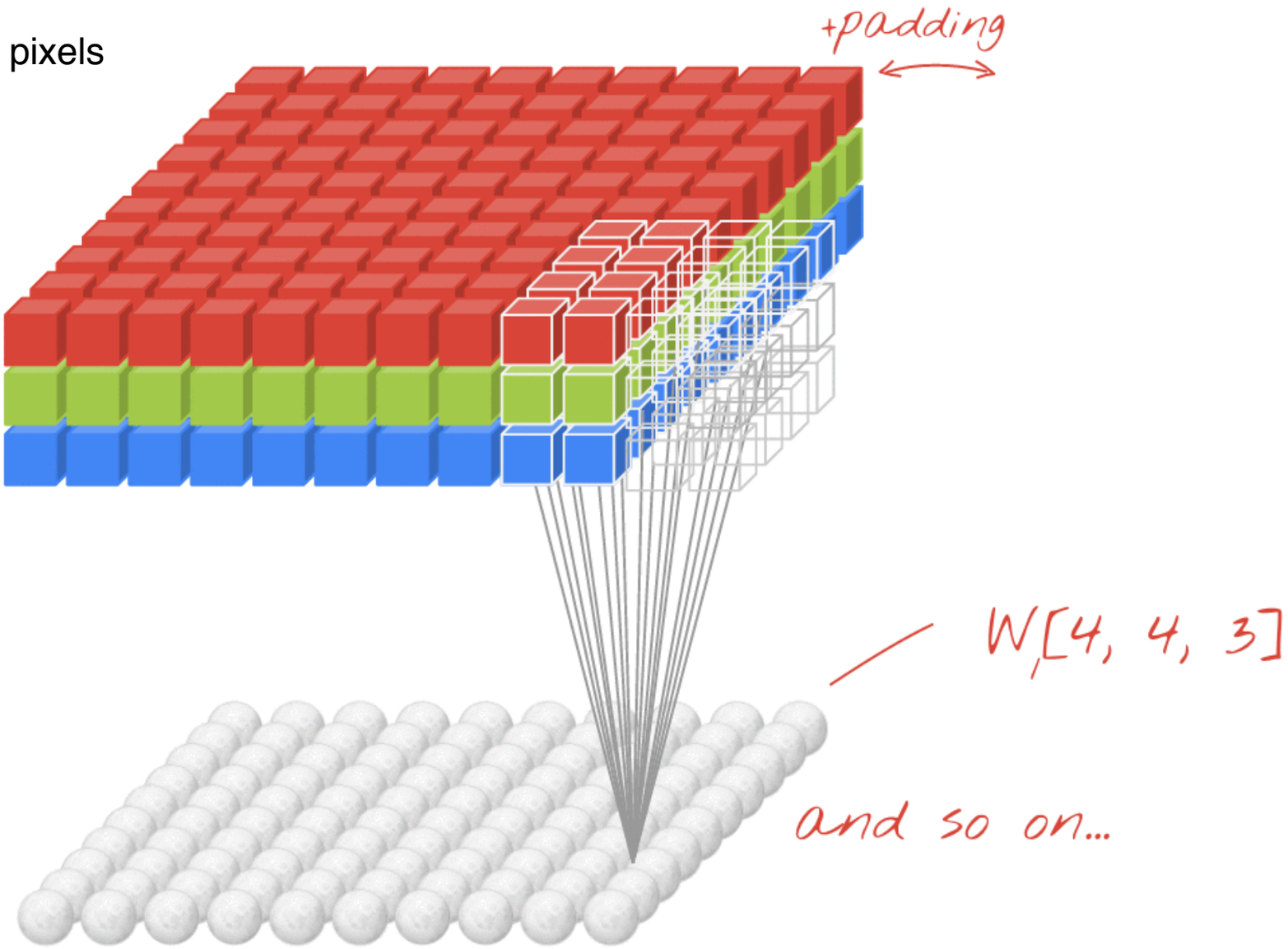
Getting up 98% accuracy: CNN

RGB pixels

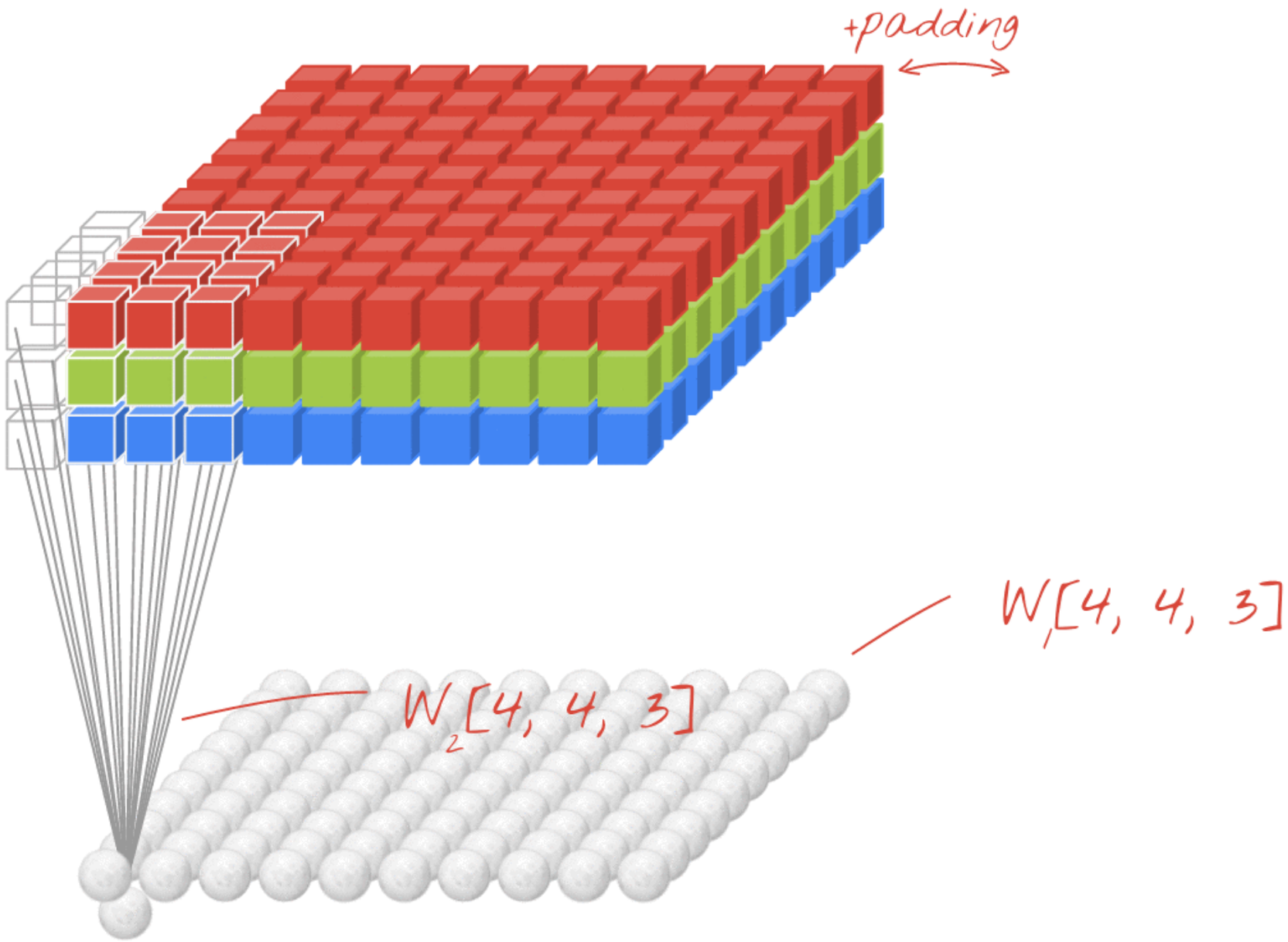


Getting up 98% accuracy: CNN

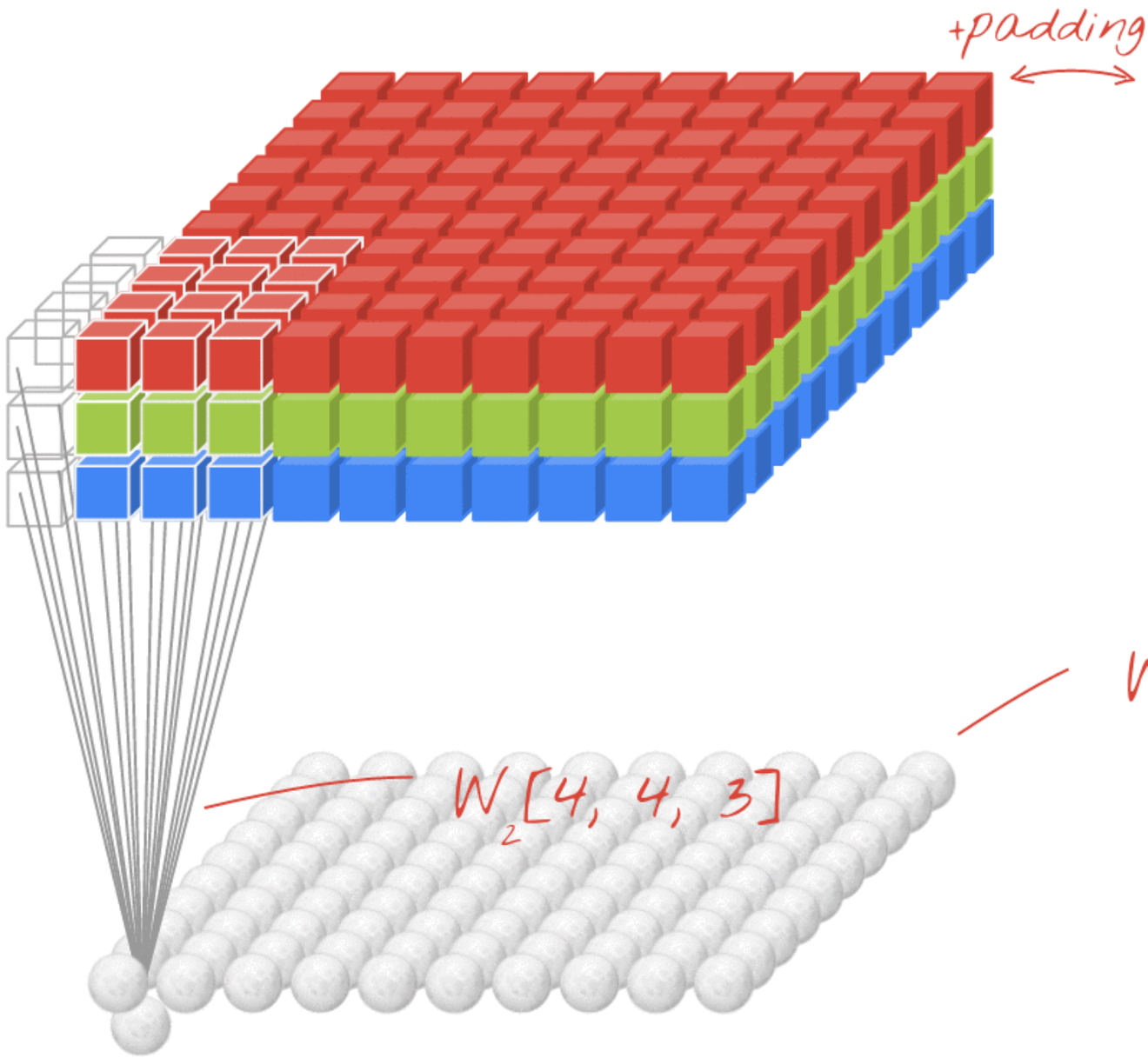
RGB pixels



Convolutional Neural Networks



Convolutional Neural Networks



shape of one convolutional layer

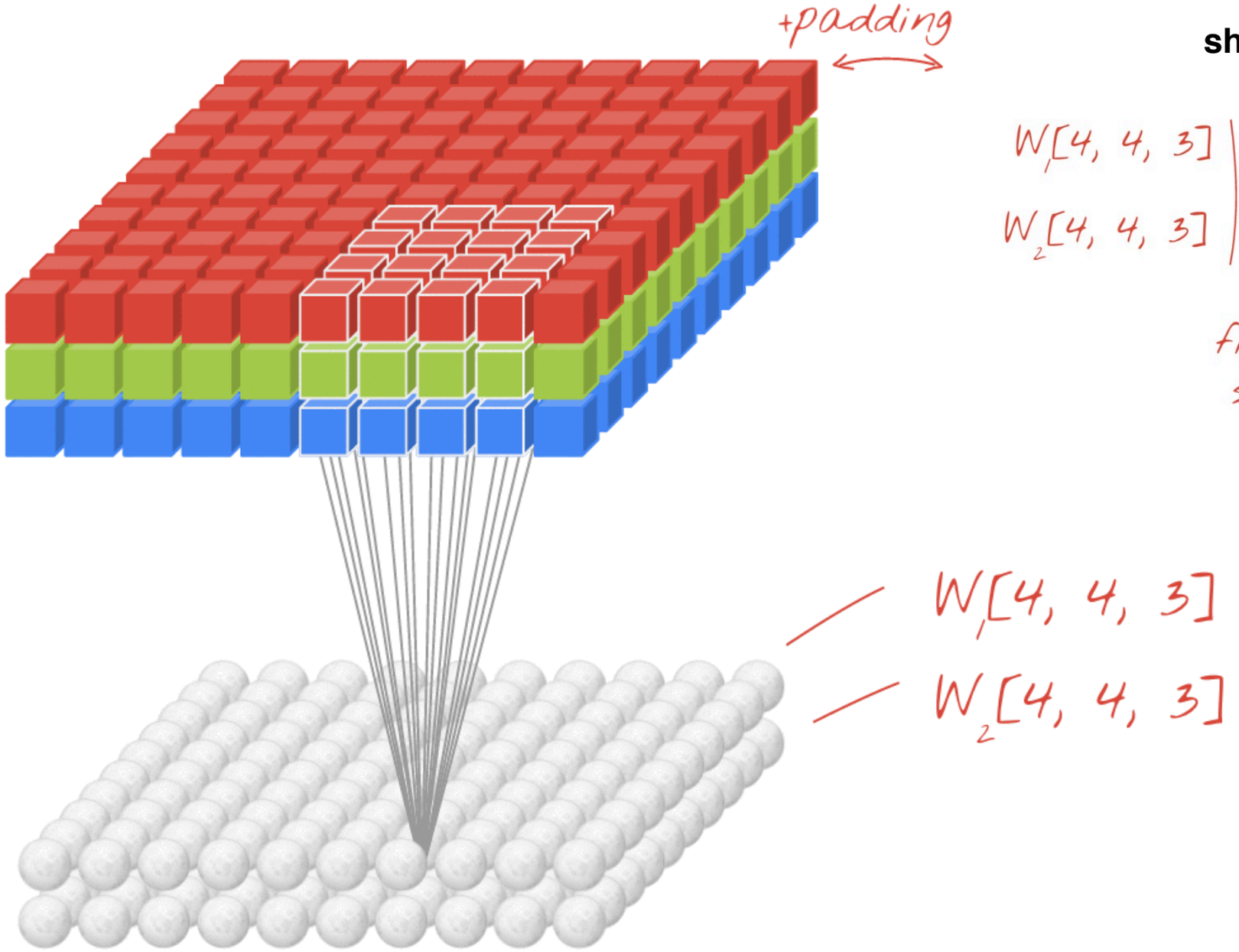
$W[4, 4, 3, 2]$

filter size

input channels

output channels

Convolutional Neural Networks



shape of one convolutional layer

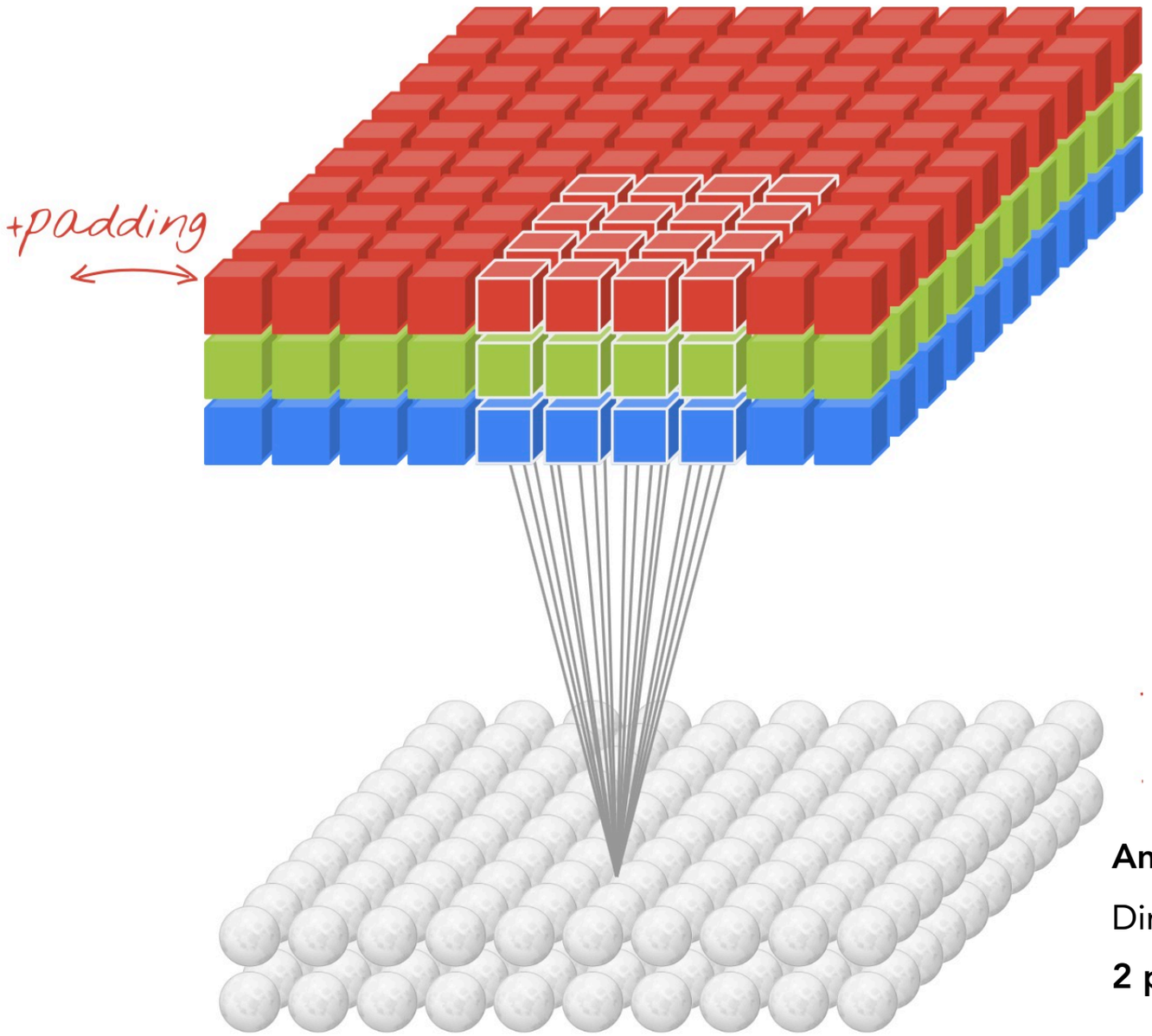
$W[4, 4, 3, 2]$

filter size

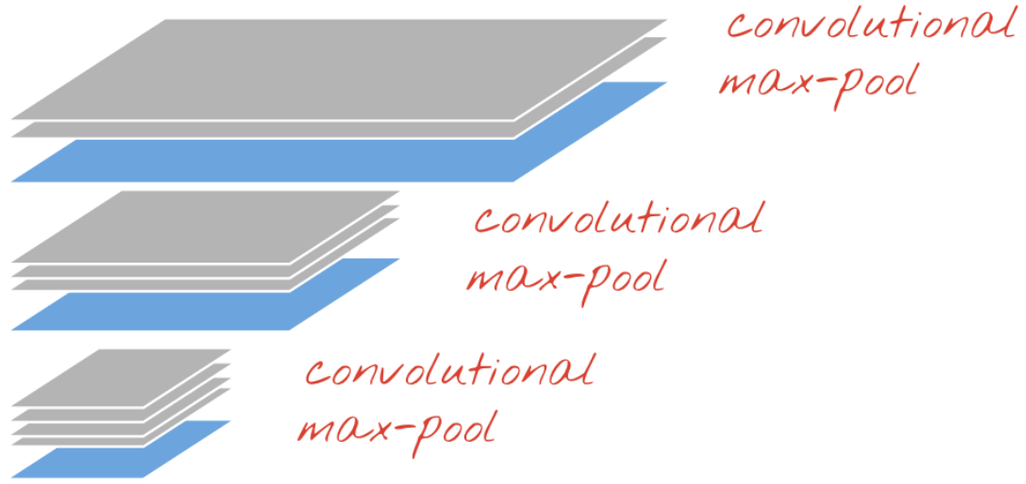
input channels

output channels

Piping information down



Traditional approach: "max pooling"

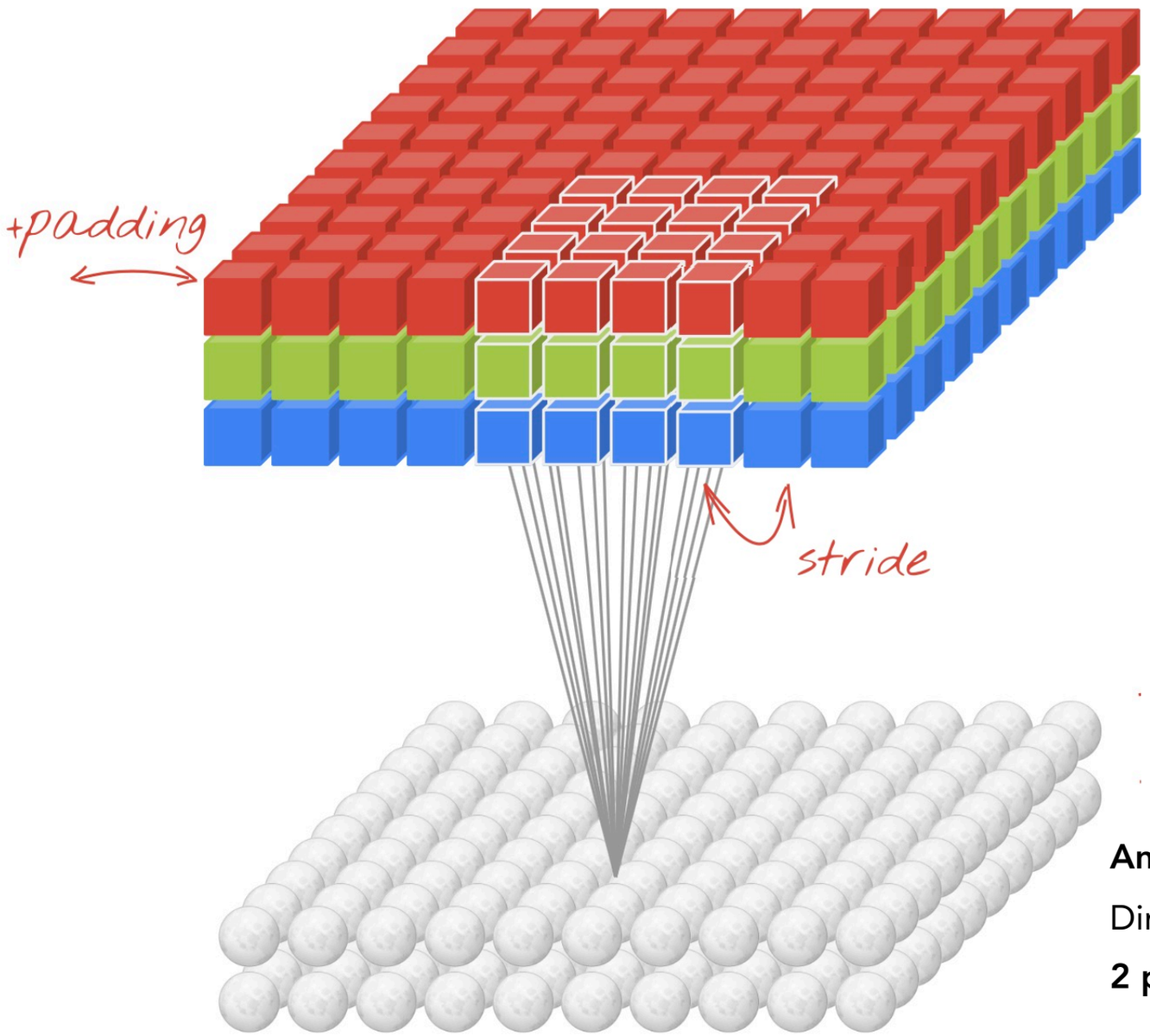


subsample the neuron outputs
by keeping only outputs where the signal is strongest

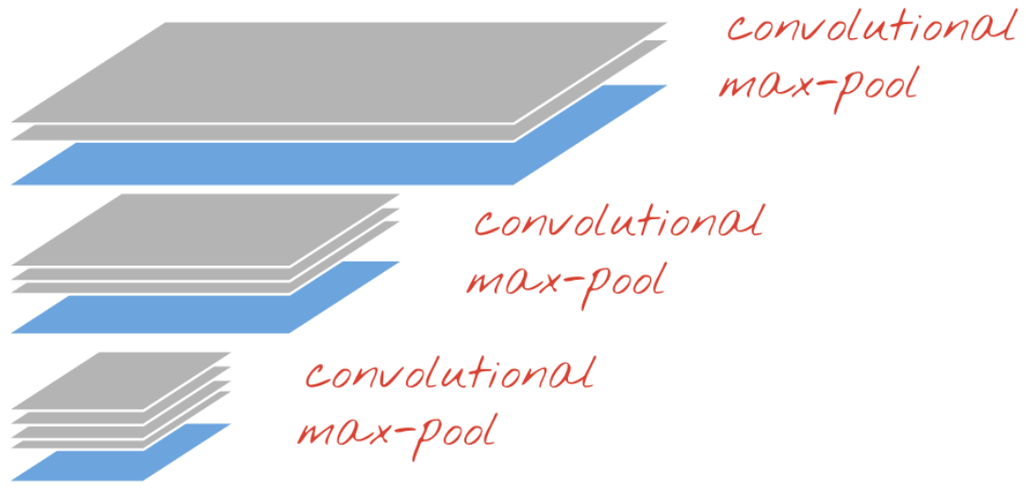
An alternative way that allows using only convolutional layers:
Directly slide the patches across the image with a **stride of 2 pixels instead of 1:** this allows to retain less outputs

This approach has proven just as effective and today's convolutional networks use convolutional layers only (see next slide)

Piping information down



Traditional approach: "max pooling"

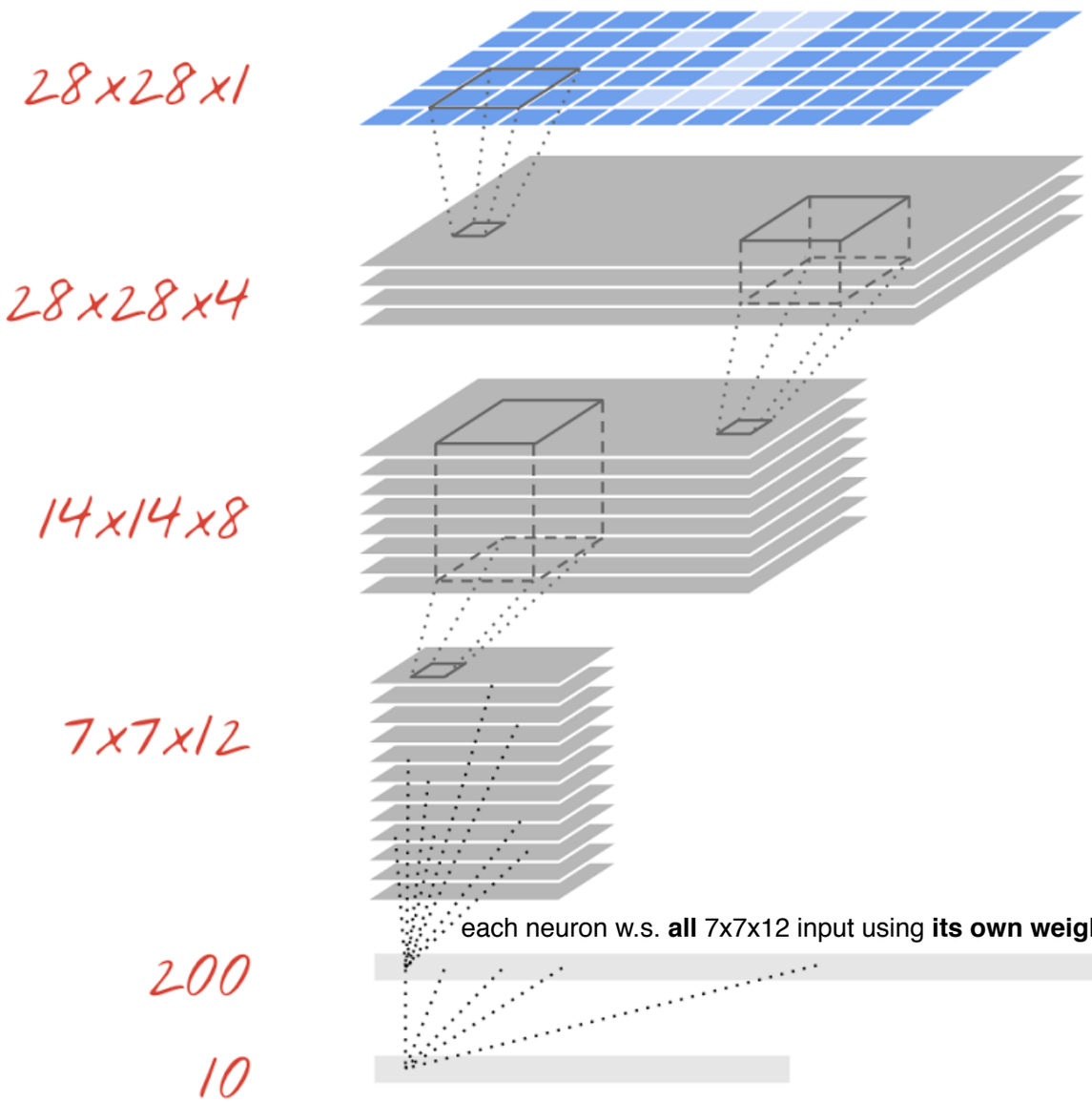


subsample the neuron outputs
by keeping only outputs where the signal is strongest

An alternative way that allows using only convolutional layers:
Directly slide the patches across the image with a **stride of 2 pixels instead of 1:** this allows to retain less outputs

This approach has proven just as effective and today's convolutional networks use convolutional layers only (see next slide)

A CNN for MNIST (all convolutional)



+ biases on all layers

convolutional layer, 4 channels
W1[5, 5, 1, 4] stride 1

convolutional layer, 8 channels
W2[4, 4, 4, 8] stride 2

convolutional layer, 12 channels
W3[4, 4, 8, 12] stride 2

fully connected layer
W4[7x7x12, 200]

softmax readout layer
W5[200, 10]

CNN in TF (Init)

filter size — input channels — output channels = number of patches we produce in output

```
K=4
L=8
M=12

W1 = tf.Variable(tf.truncated_normal([5, 5, 1, K] ,stddev=0.1))
B1 = tf.Variable(tf.ones([K])/10)
W2 = tf.Variable(tf.truncated_normal([5, 5, K, L] ,stddev=0.1))
B2 = tf.Variable(tf.ones([L])/10)
W3 = tf.Variable(tf.truncated_normal([4, 4, L, M] ,stddev=0.1))
B3 = tf.Variable(tf.ones([M])/10)

N=200

W4 = tf.Variable(tf.truncated_normal([7*7*M, N] ,stddev=0.1))
B4 = tf.Variable(tf.ones([N])/10)
W5 = tf.Variable(tf.truncated_normal([N, 10] ,stddev=0.1))
B5 = tf.Variable(tf.zeros([10])/10)
```

weights initialised with random values

CNN in TF (Model)

conv2d: you give it batch of input values (images), a set of weights, it will scan input in both directions and produce the weighted sums

input image batch
X[100, 28, 28, 1]

weights

stride

biases

activation functions

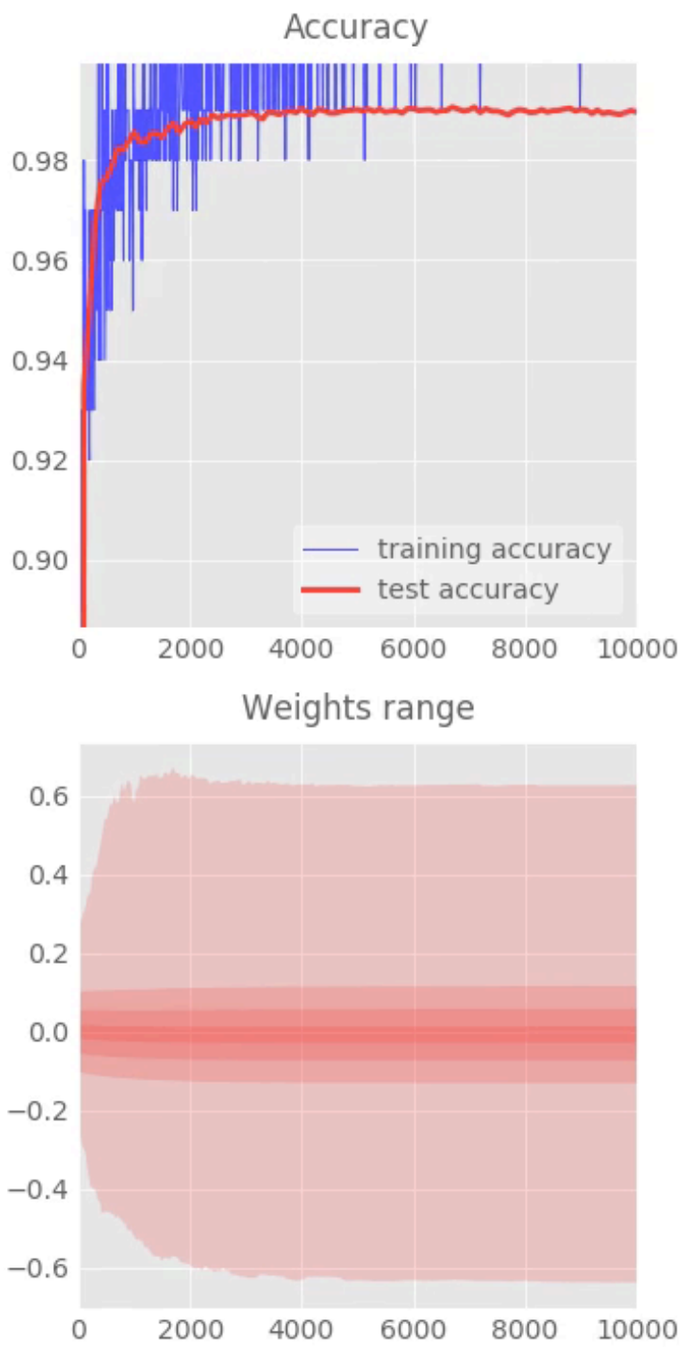
```
Y1 = tf.nn.relu(tf.nn.conv2d(X, W1, strides=[1, 1, 1, 1], padding='SAME') + B1)
Y2 = tf.nn.relu(tf.nn.conv2d(Y1, W2, strides=[1, 2, 2, 1], padding='SAME') + B2)
Y3 = tf.nn.relu(tf.nn.conv2d(Y2, W3, strides=[1, 2, 2, 1], padding='SAME') + B3)

YY = tf.reshape(Y3, shape=[-1, 7 * 7 * M])

Y4 = tf.nn.relu(tf.matmul(YY, W4) + B4)
Y  = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

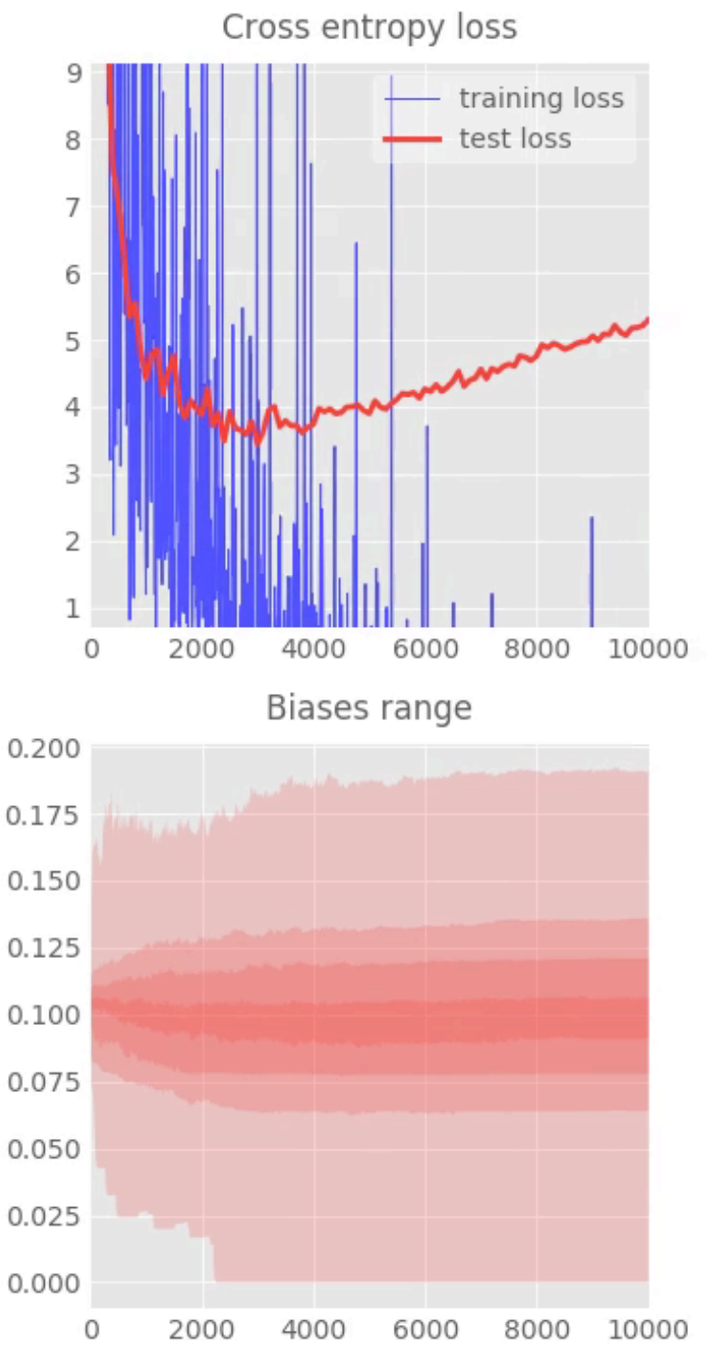
flatten all values for fully connected layer

Y3 [100, 7, 7, 12]

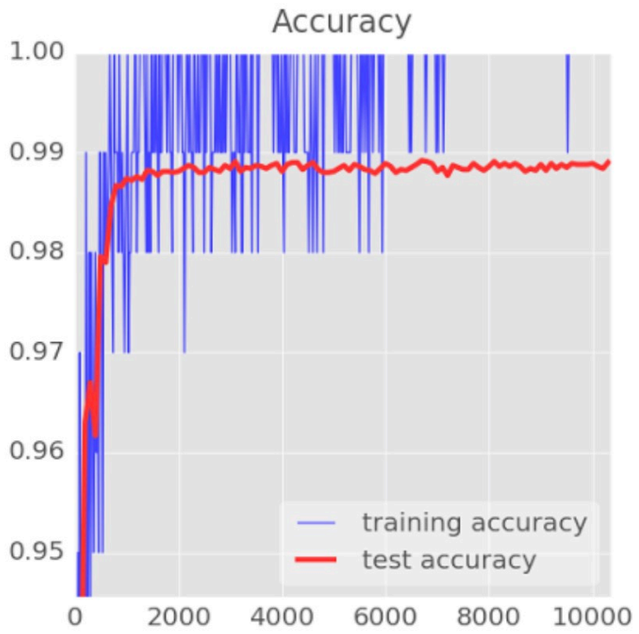
YY [100, 7x7x12]

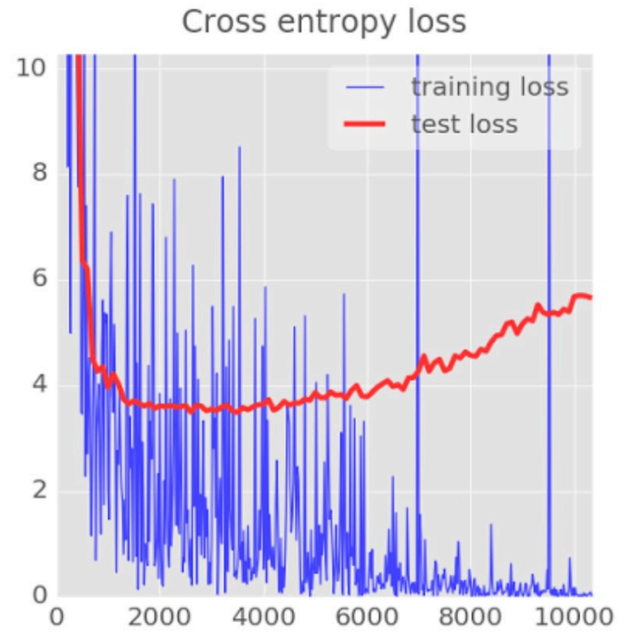
Demo: mnist_3.0_convolutional.py



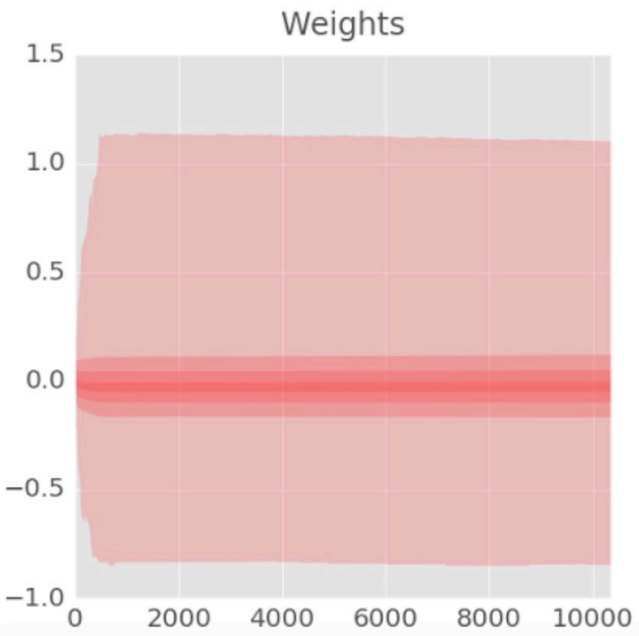


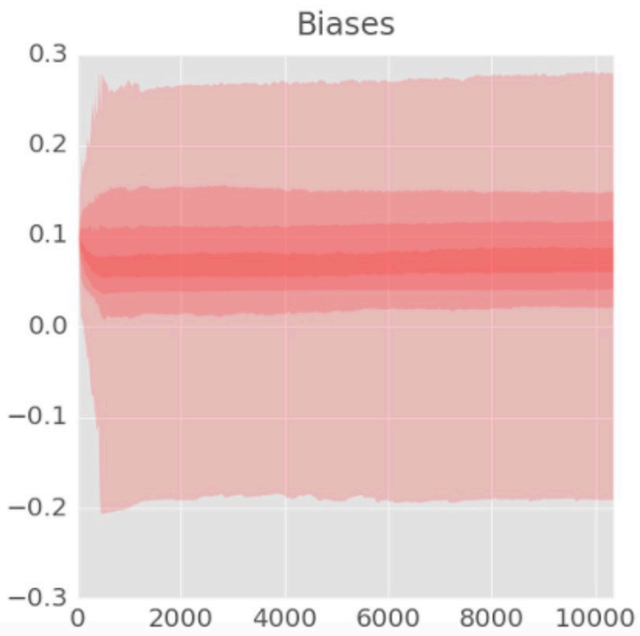
Results -CNN

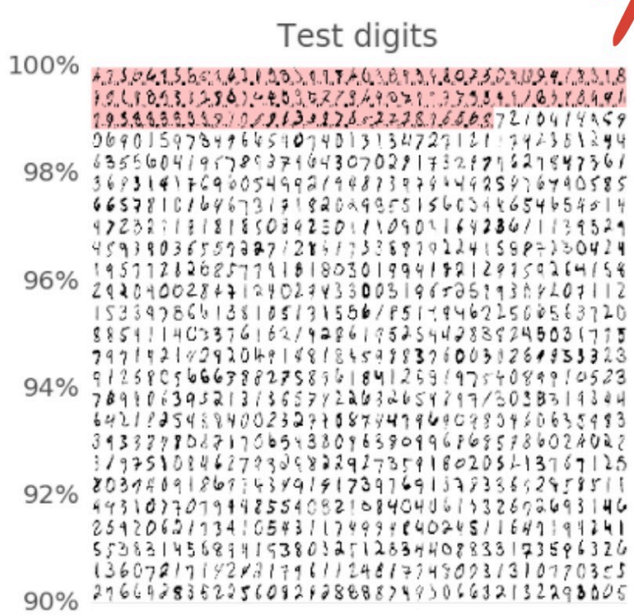




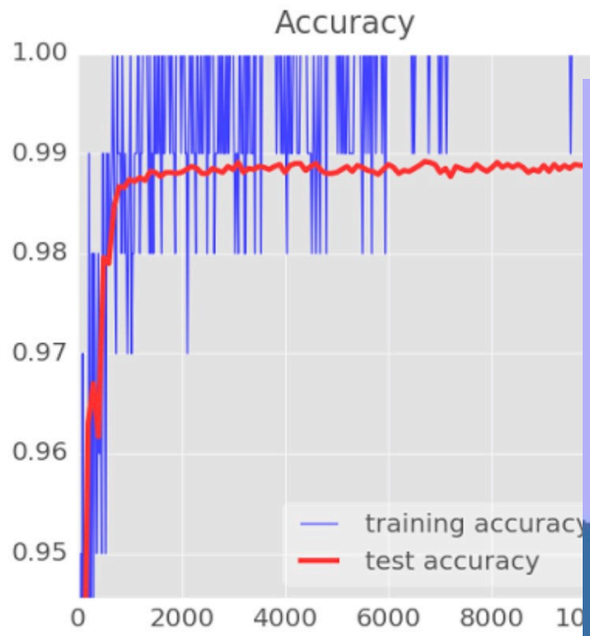
98.9%

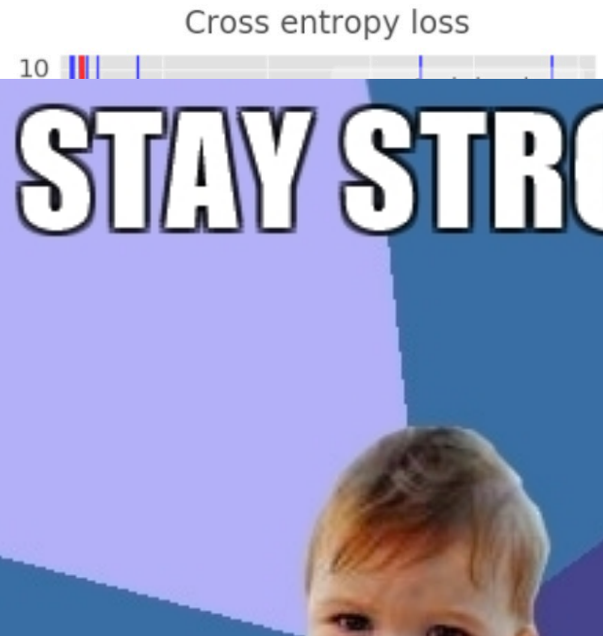


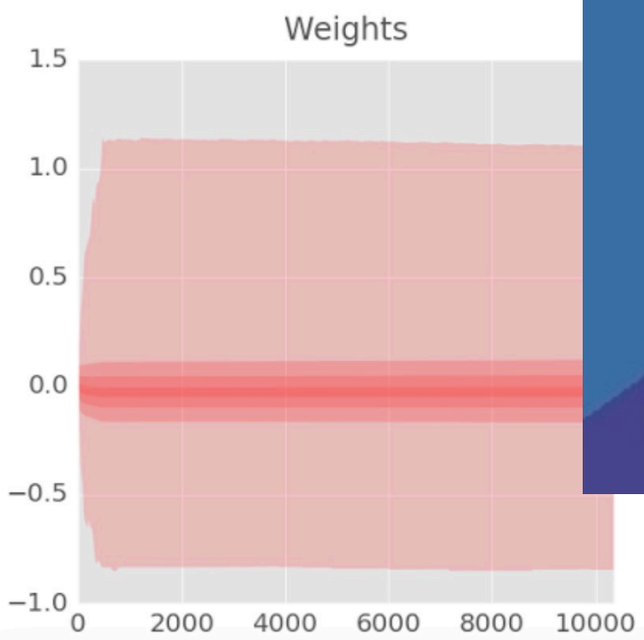




Results -CNN

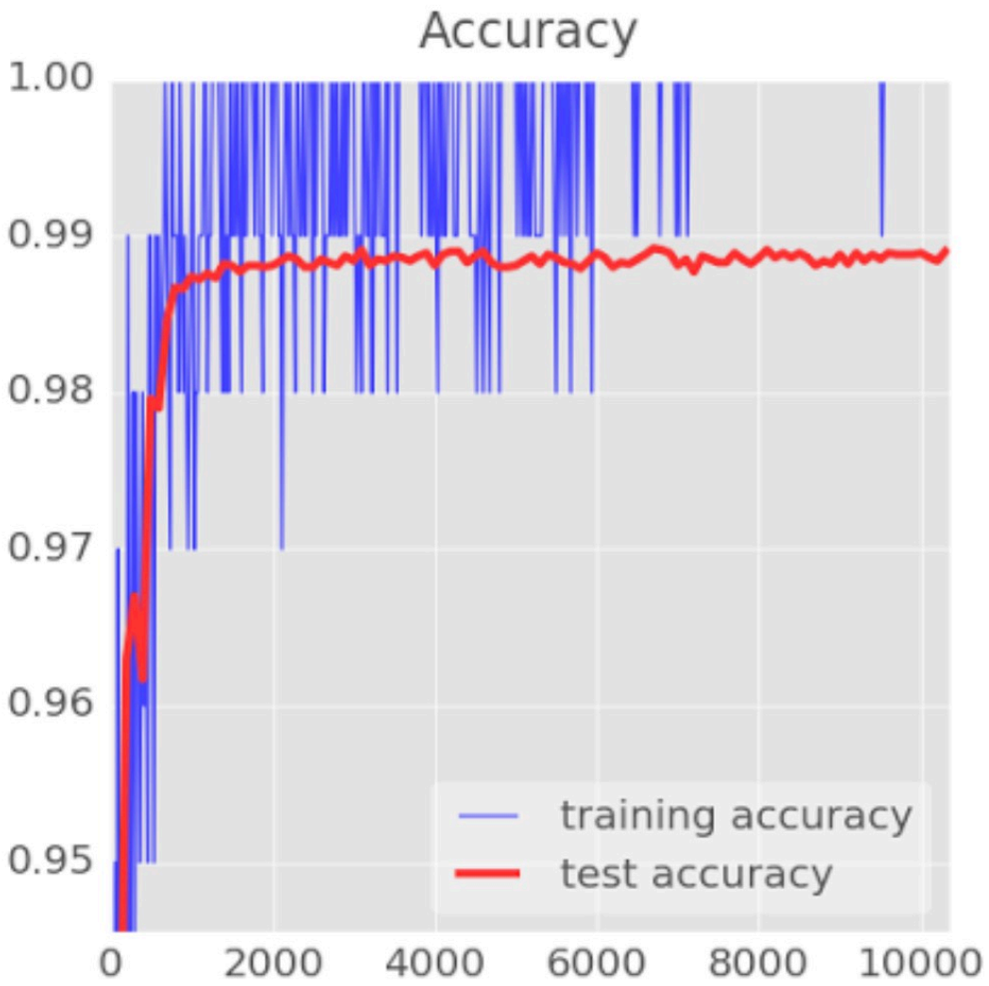


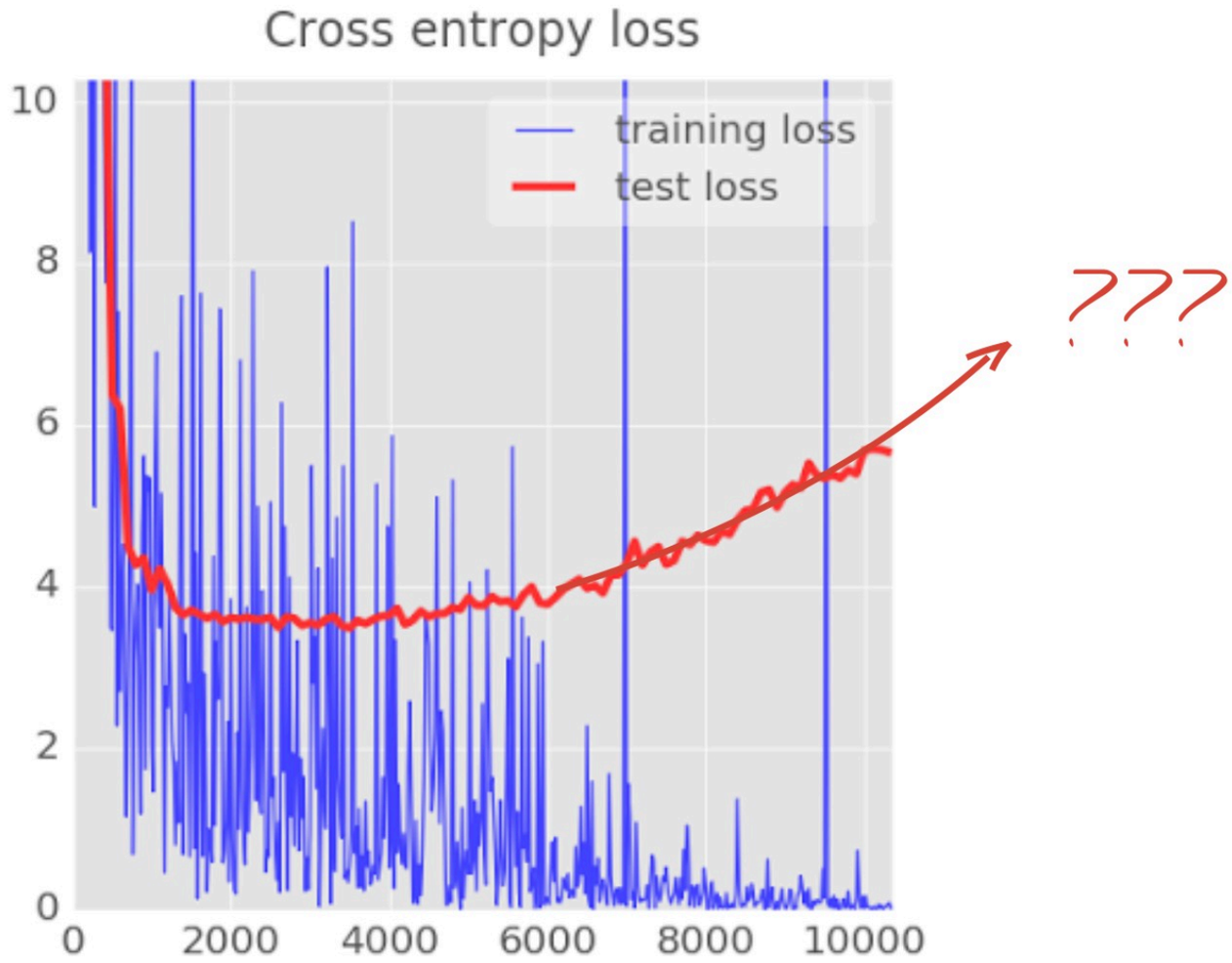




98.9%

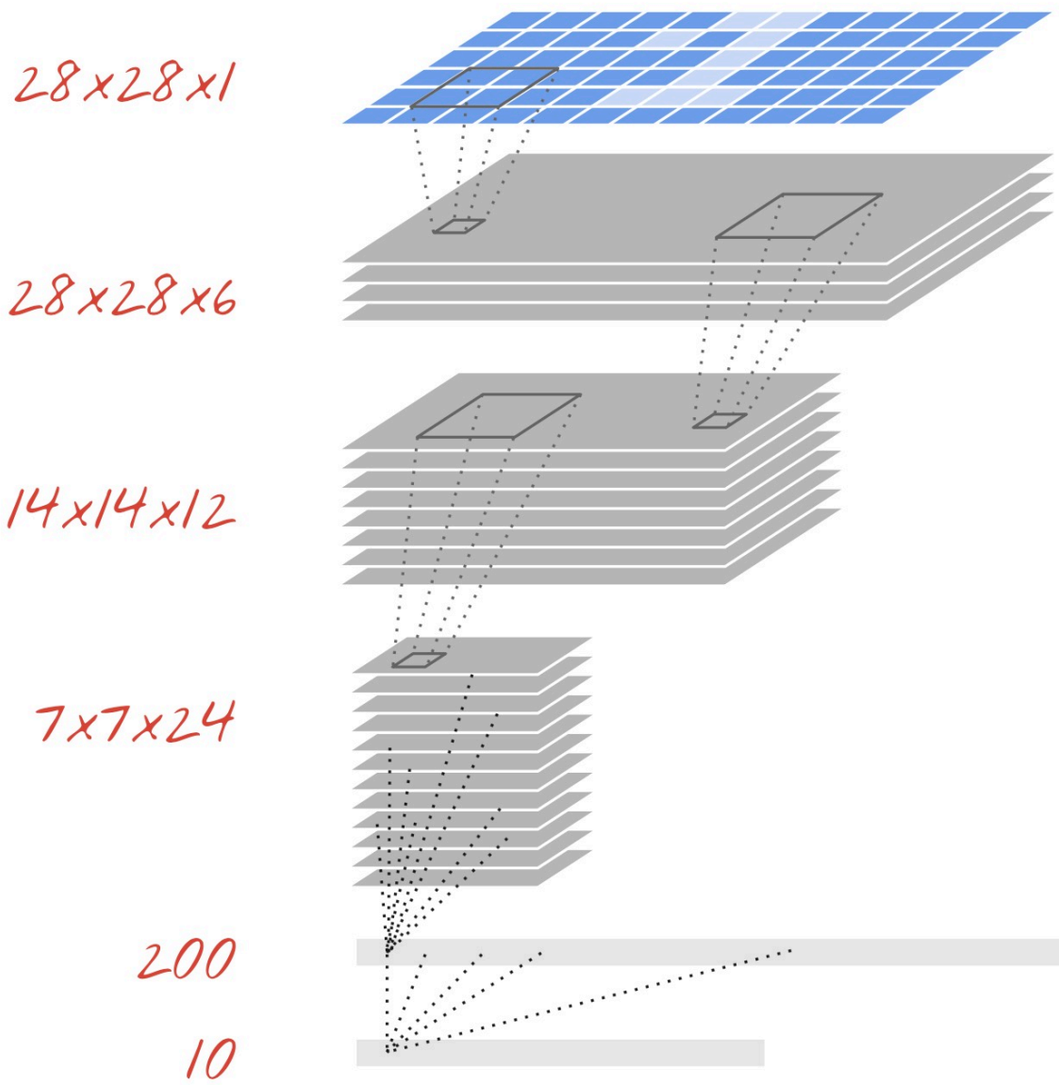
WTFH?





Solutions?

Go bigger



+ biases on all layers

convolutional layer, 6 channels
W1[6, 6, 1, 6] stride 1

patch size

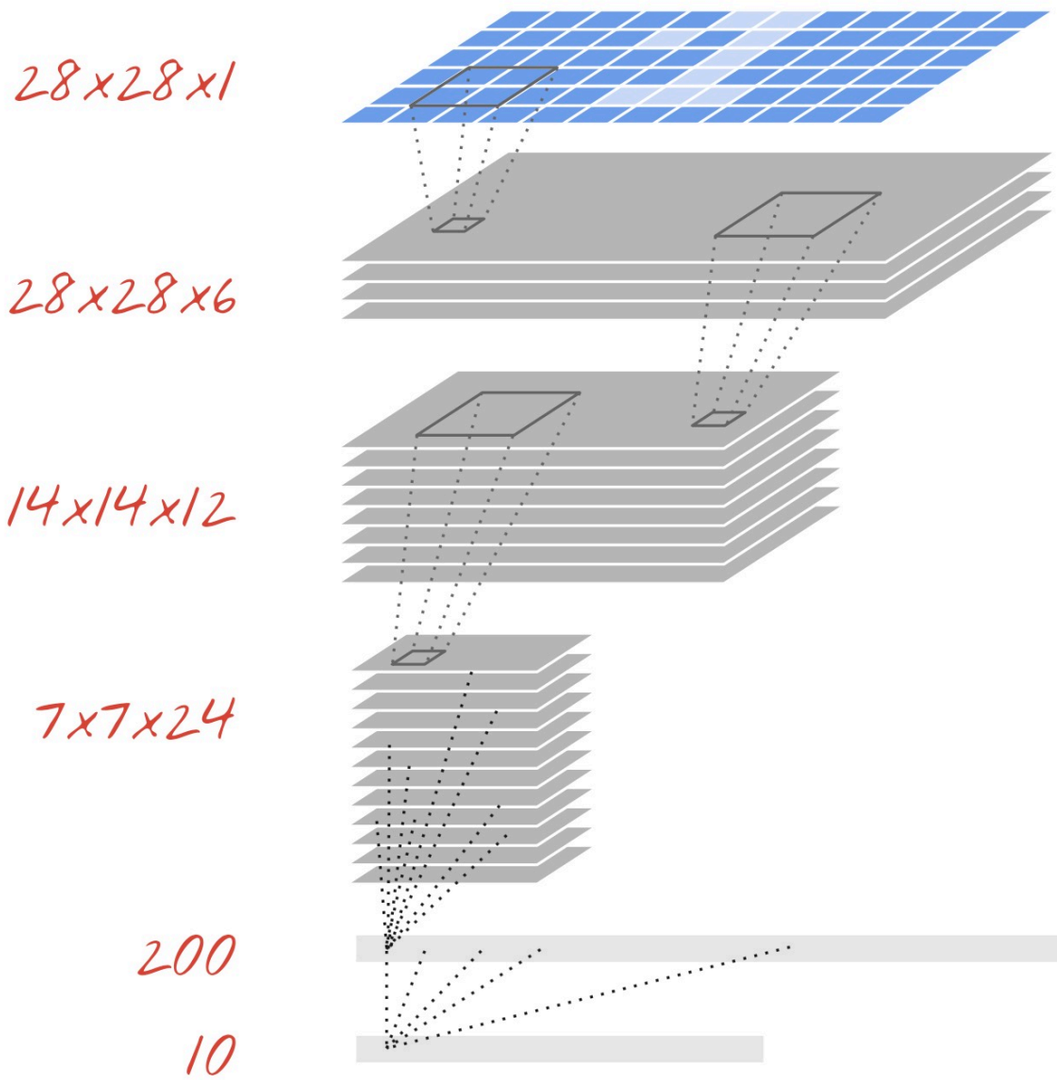
convolutional layer, 12 channels
W2[5, 5, 6, 12] stride 2

convolutional layer, 24 channels
W3[4, 4, 12, 24] stride 2

fully connected layer W4[7x7x24, 200]
softmax readout layer W5[200, 10]

why not apply dropout to previous layers?

Go bigger



+ biases on all layers

convolutional layer, **6 channels**
W1[**6, 6**, 1, 6] stride 1

patch size

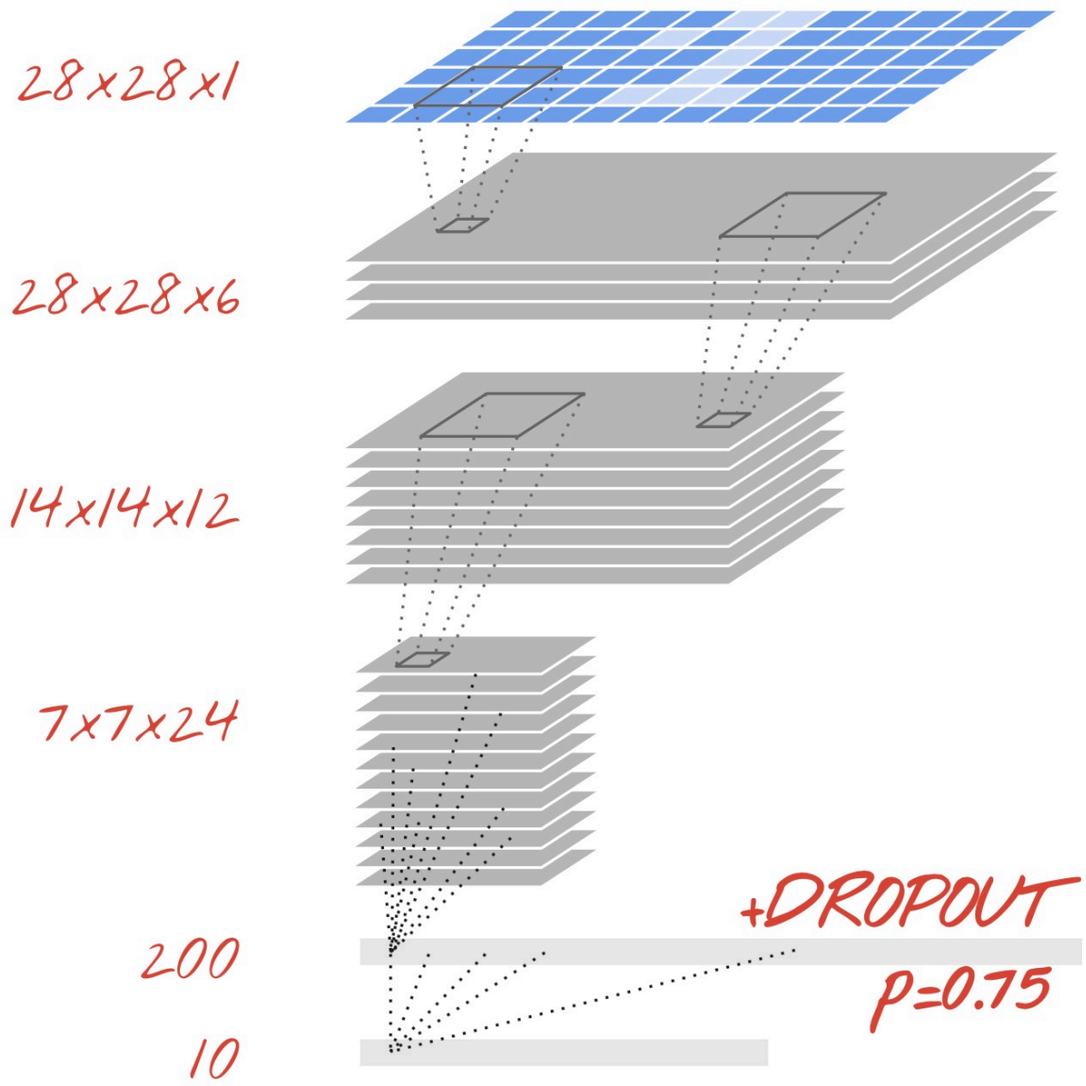
convolutional layer, **12 channels**
W2[**5, 5**, 6, 12] stride 2

convolutional layer, **24 channels**
W3[**4, 4**, 12, 24] stride 2

fully connected layer W4[7x7x24, 200]
softmax readout layer W5[200, 10]

why not apply dropout to previous layers?

Go bigger



+ biases on all layers

convolutional layer, **6 channels**
W1[**6, 6**, 1, 6] stride 1
patch size

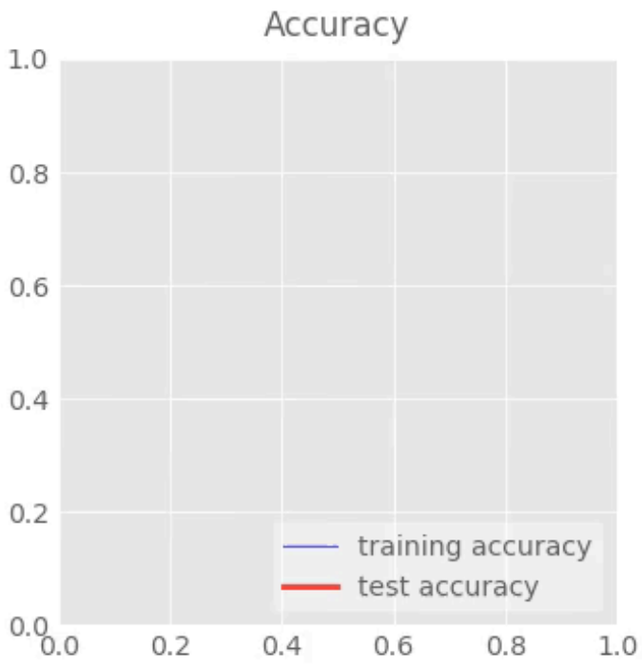
convolutional layer, **12 channels**
W2[**5, 5**, 6, 12] stride 2

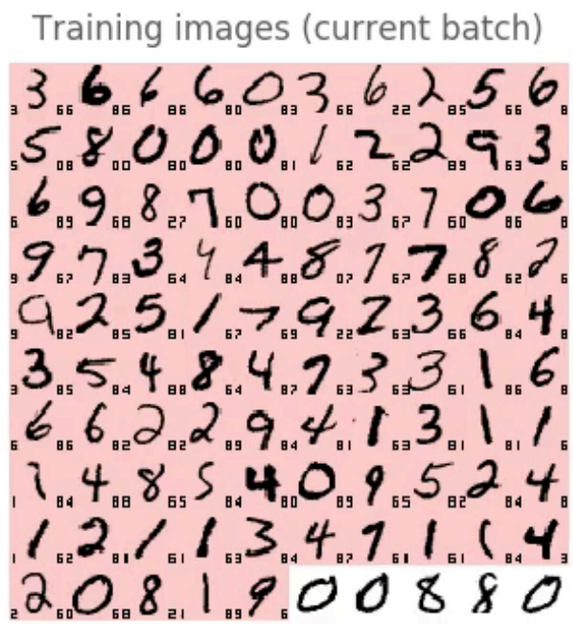
convolutional layer, **24 channels**
W3[**4, 4**, 12, 24] stride 2

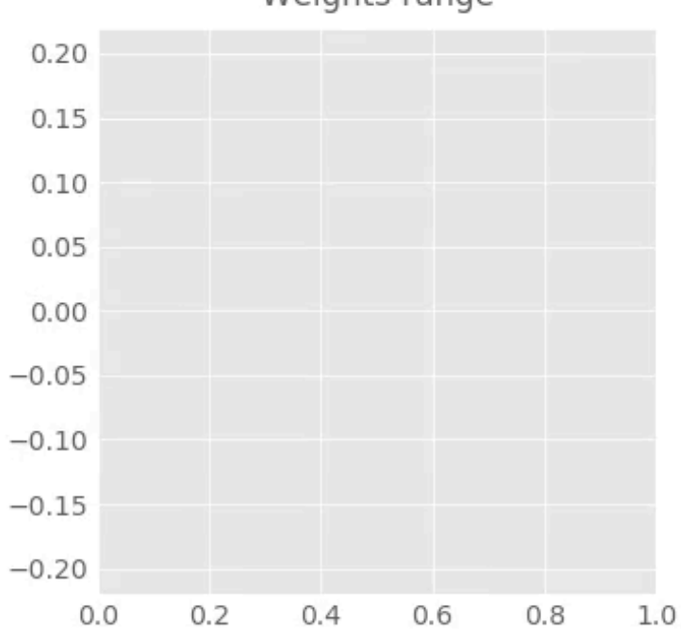
fully connected layer W4[7x7x24, 200]
softmax readout layer W5[200, 10]

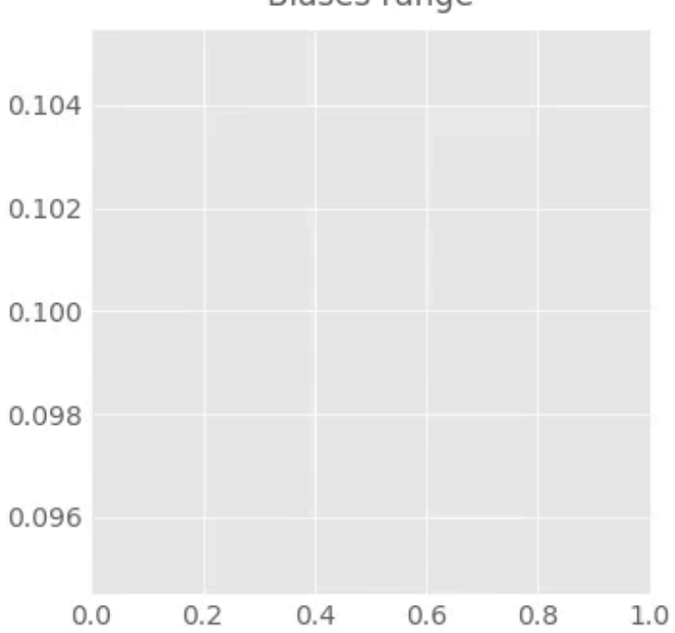
why not apply dropout to previous layers?

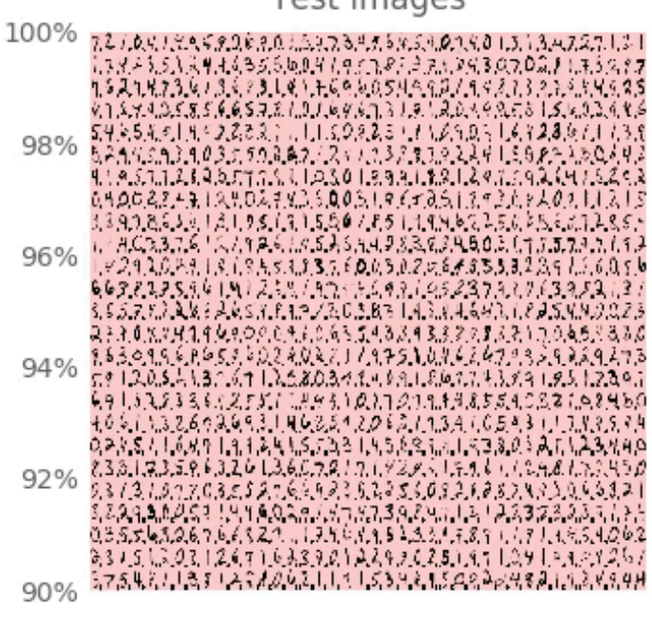
Demo: `mnist_3.1_convolutional_bigger_dropout.py`



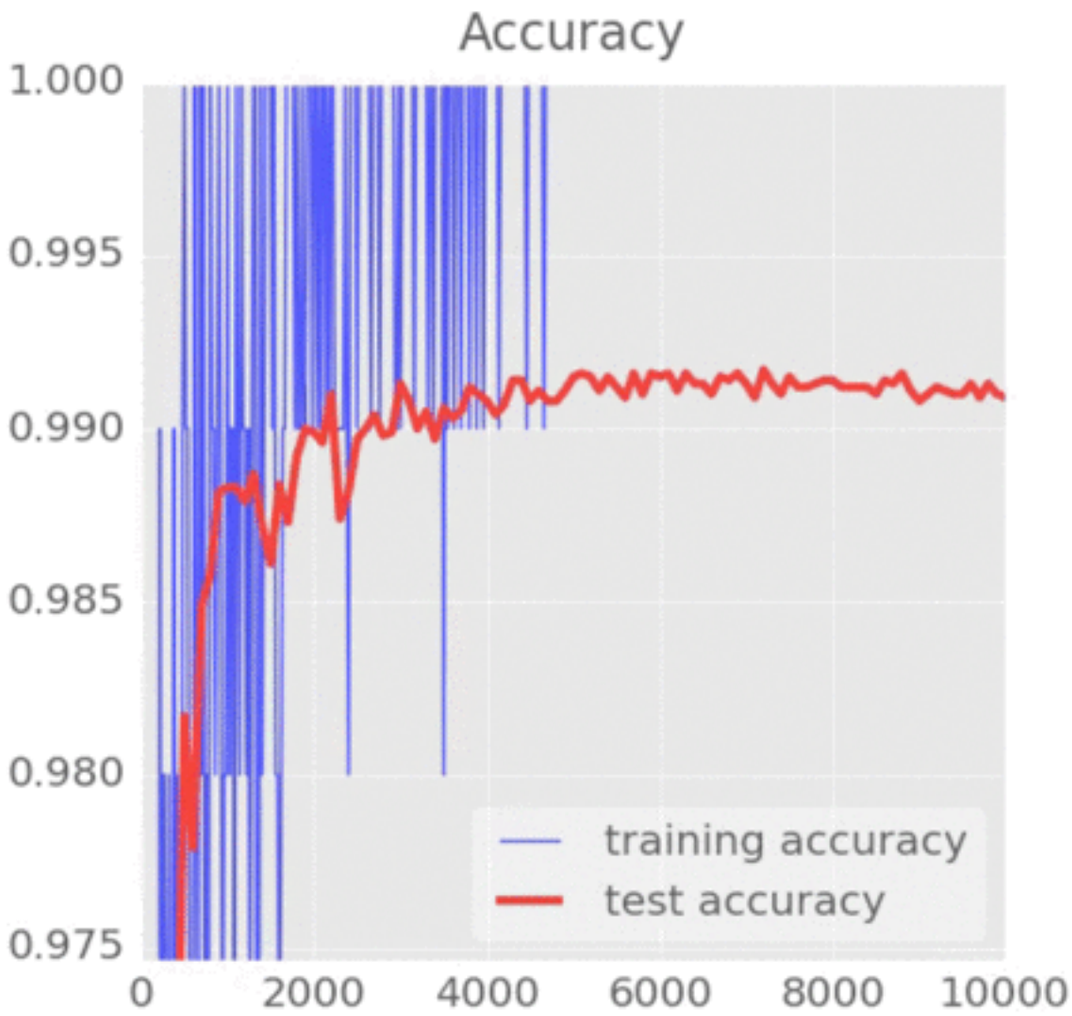






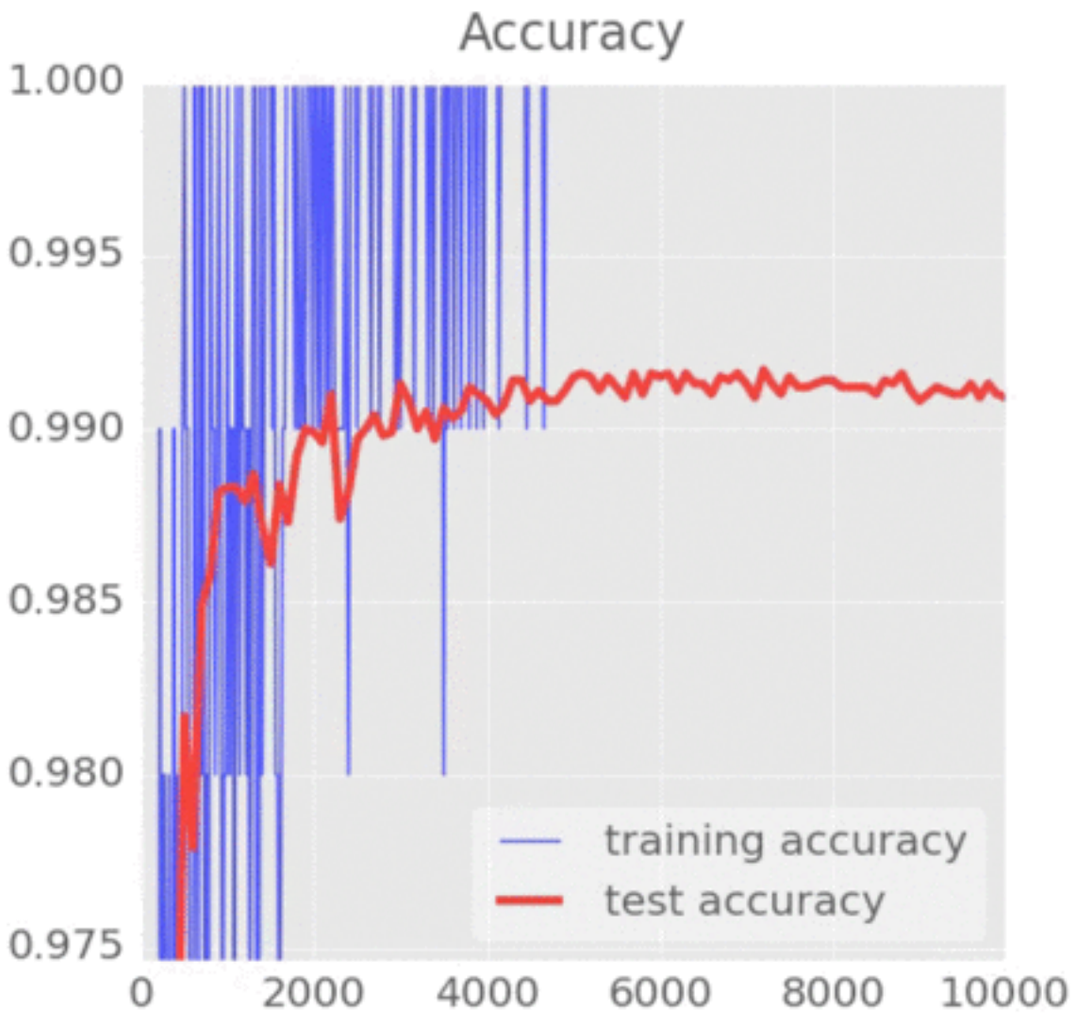


Improvements



larger convolutional network

Improvements



larger convolutional network

Success?

Our model now misses only **72** out of the **10,000** test digits, with 99.3% accuracy

The world record, which you can find on the MNIST website is around **99.7%**.
We are only 0.4 percentage points away from it with our model built with 100 lines of Python / TensorFlow.

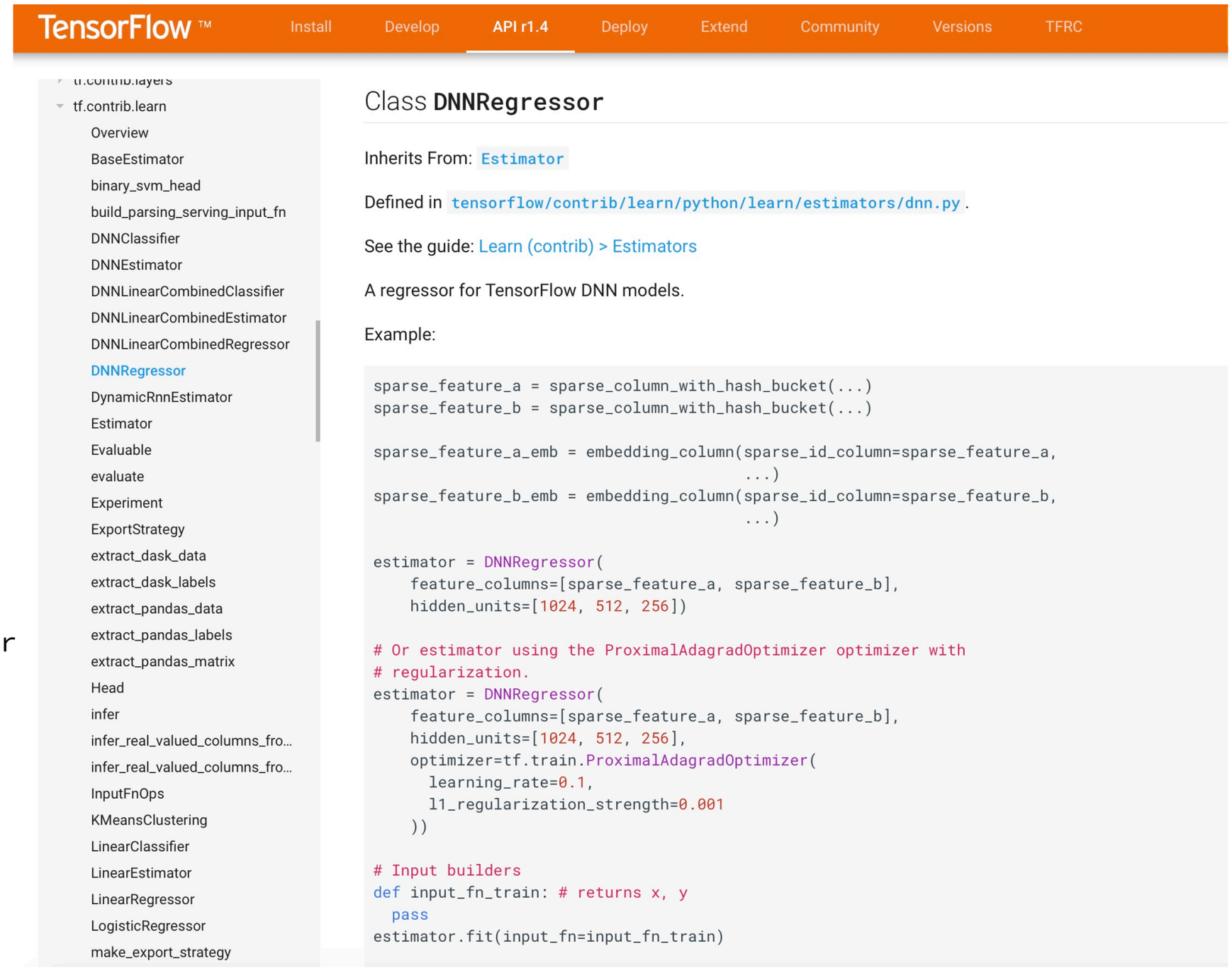
Another technique that we didn't mention is **batch normalization**, that can lead us up to 99.5 accuracy.

TF.Learn, TF High level APIs

Tensorflow higher-level APIs too called tf.learn.

Simple but constrained.

Example: `DNNRegressor`



References and pointers

Code

```
git clone https://github.com/maruscia/tensorflow-mnist-tutorial
```

- largely (and aggressively) borrowed by work by **Martin Görner**
  - https://cloud.google.com/blog/big-data/2017/01/learn-tensorflow-and-deep-learning-without-a-phd

- **Deep Learning:** Advanced machine learning course on neural networks, with extensive coverage of image and text models
- **Rules of ML:** Best practices for machine learning engineering
- **deeplearn.js:** Open-source toolkit for interactive model training and inference in the browser
- **distill.pub:** Web journal presenting recent research in ML & AI in a divulgative way (reactive diagrams, visualizations, mind diagrams)
- **colah.github.io:** Blog by **Chris Olah**, (Google Brain team, also editor at distill)
  - In particular check out this blog post for (quite old, yet still very informative) visualizations of the MNIST

TensorFlow resources

- **TensorFlow Programmer's Guide**: In-depth guides to key TensorFlow features, including variables, threading, and debugging
- **TensorFlow Dev Summit 2017**: Series of tech talks and demos highlighting TensorFlow APIs and real-world applications