Learning Systems 2018:
Lecture 5 – Gradient Descent

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens

Argonne National Laboratory

The University of Chicago

```python
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```python
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```python
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

```python
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```python
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

layer_1 = W1 * X + B1
layer_2 = W2 * layer_1 + B2
out_layer = Wout * layer_2 + Bout
output = softmax(out_layer)

Note: This simplified network only has an activation function at the final layer, rather than one per layer.

MNIST in Tensorflow

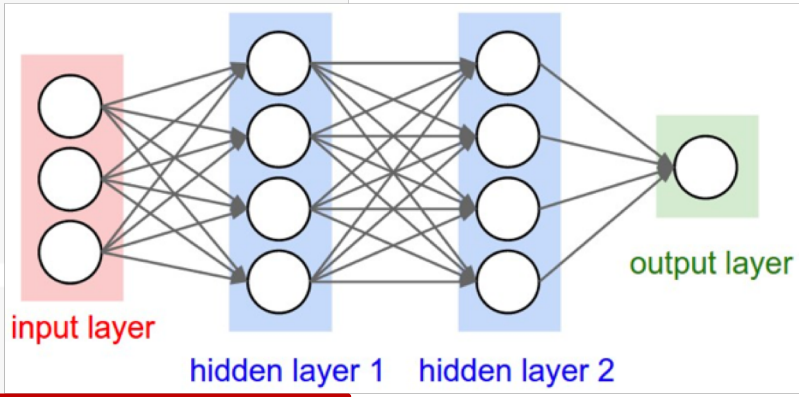
```
# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
```

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

layer_1 = W1 * X + B1

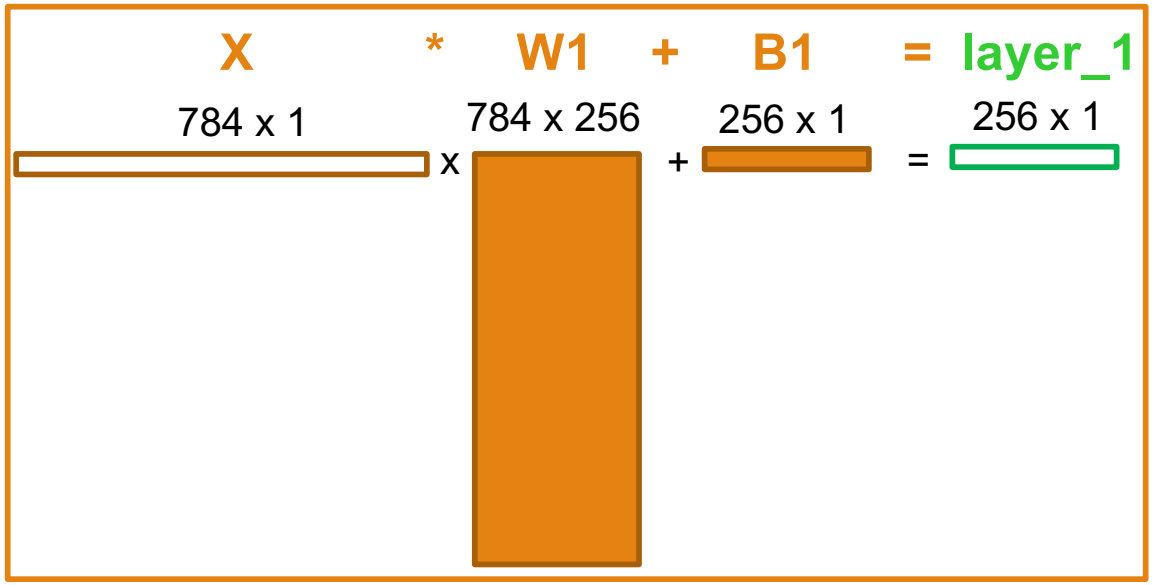
layer_2 = W2 * layer_1 + B2

out_layer = Wout * layer_2 + Bout

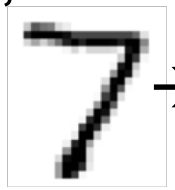



```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

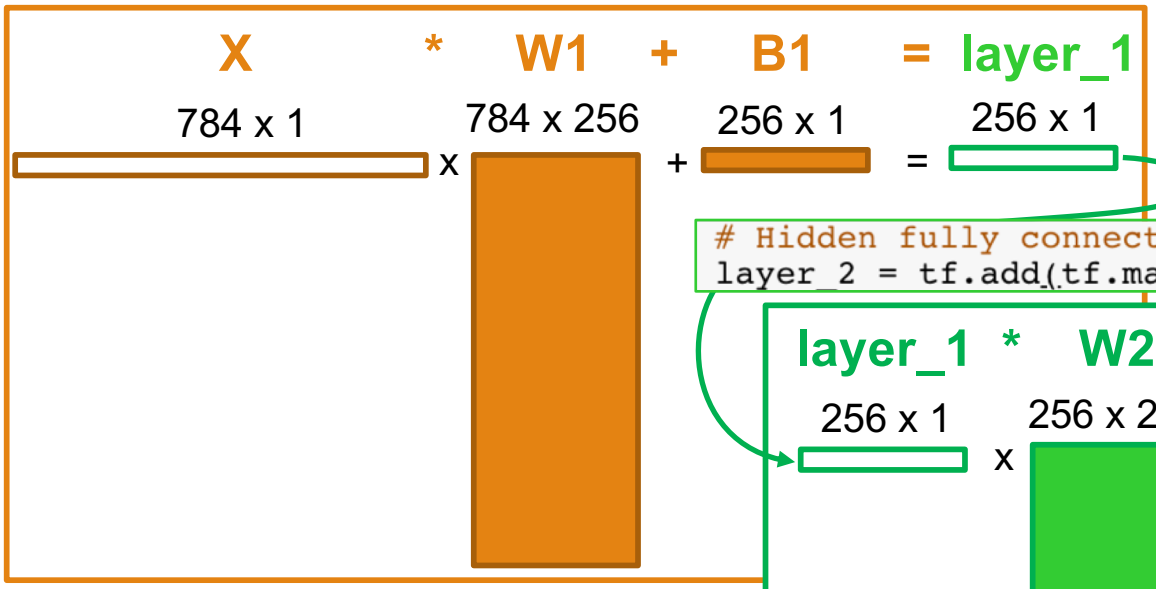
```
# Hidden fully connected layer with 256 neurons
layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
```



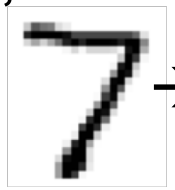
One forward pass: Inference
E.g., data → label



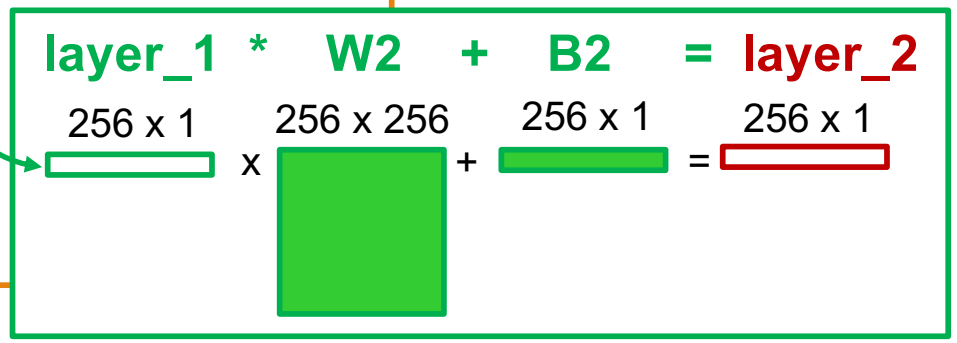
```
# Hidden fully connected layer with 256 neurons
layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
```



One forward pass: Inference
E.g., data → label




```
# Hidden fully connected layer with 256 neurons
layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
```

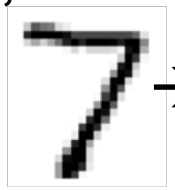


```
# Hidden fully connected layer with 256 neurons
layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
```

X * W1 + B1 = layer_1

784 x 1 x 784 x 256 + 256 x 1 = 256 x 1

One forward pass: Inference
E.g., data → label



→ 7

```
# Hidden fully connected layer with 256 neurons
layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
```

layer_1 * W2 + B2 = layer_2

256 x 1 x 256 x 256 + 256 x 1 = 256 x 1

```
# Output fully connected layer with a neuron for each class
out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
```

layer_2 * Wout + Bout = out

256 x 1 x 256 x 10 + 10 x 1 = 10 x 1

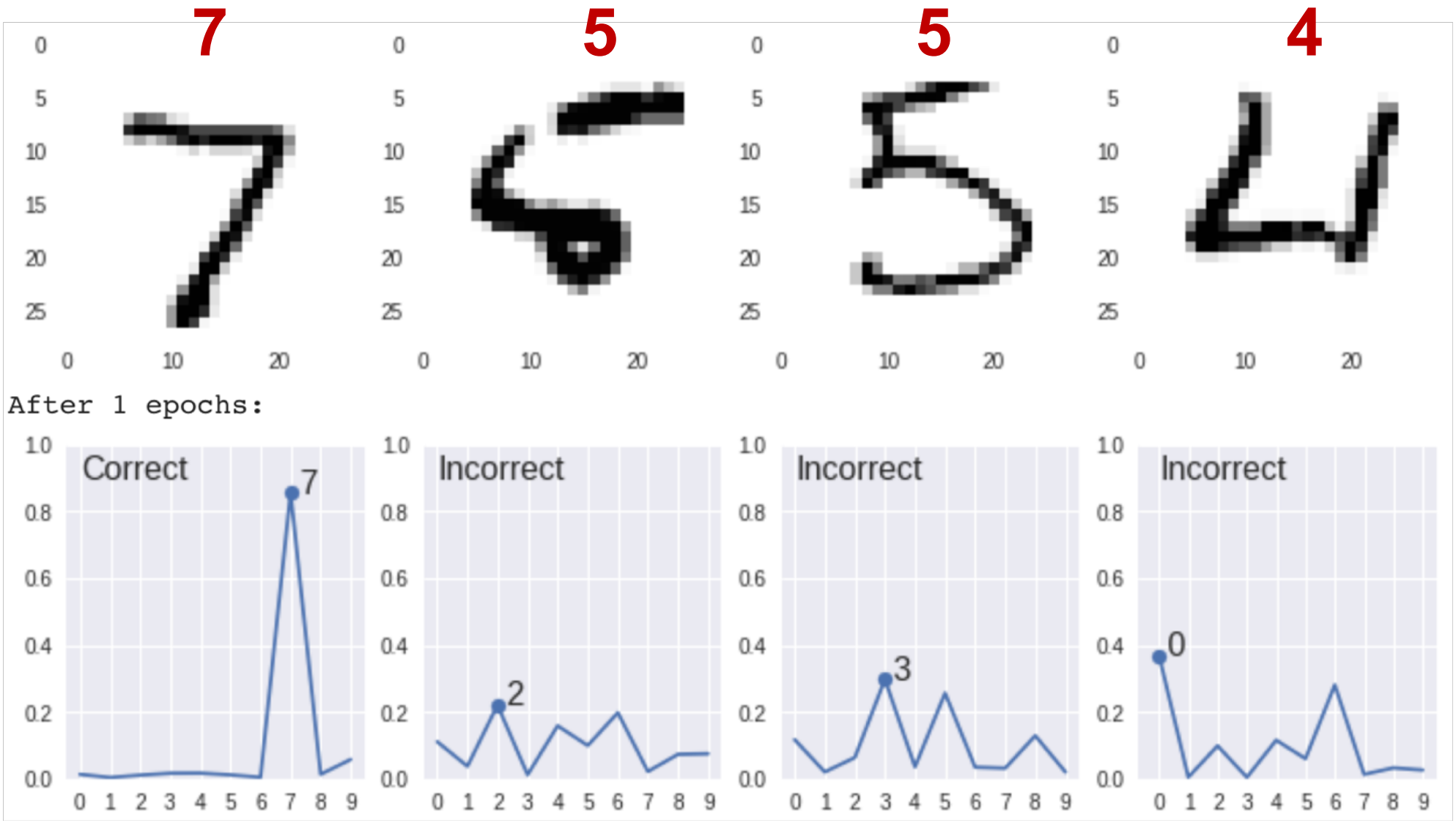
$$\mathbf{y'} = \text{softmax}\,[W_{out}\,(\,W_2\,(W_1\,\mathbf{x} + B_1) + B_2) + B_{out}]$$

Consider an even simpler (1 layer) model

```
# tf Graph Input
x = tf.placeholder(tf.float32, [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder(tf.float32, [None, 10])   # 0-9 digits recognition => 10 classes

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
nn_output = tf.matmul(x, W) + b
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
```

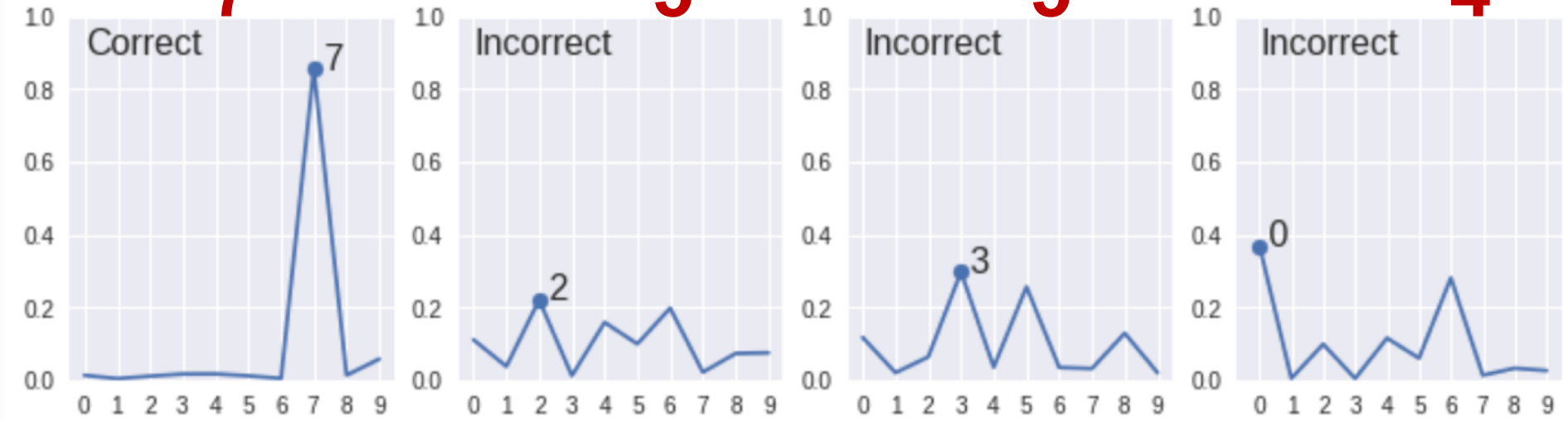
Epoch **Correct (-) or incorrect (F) on first 100 test images**

```
00: FFF-FFFFFF-FF-FFFFFFFFFF-FF-FFFFFFFFFFFFFFFFFFFFFFFFF-FFFFFFFFFFFF-F-FFFFFFFFFFFFFFFFFFFFFFFFF
01: --------F------F----F-----------F----F------FF------FF--------F-FF------F---FF-F----------F----F-
02: --------F----------------------F----F-------F-------F--------F-F-------F---F--F----------F----F-
04: --------F----------------------F----F-------F-------F--------F---------F---F--F----------F------
08: --------F----------------------F-----------F---------------F-------------F--F----------F------
16: --------F----------------------F-----------F---------------F-------------F-------------F------
32: --------F----------------------F---------------------------F-------------F-------------F------
```

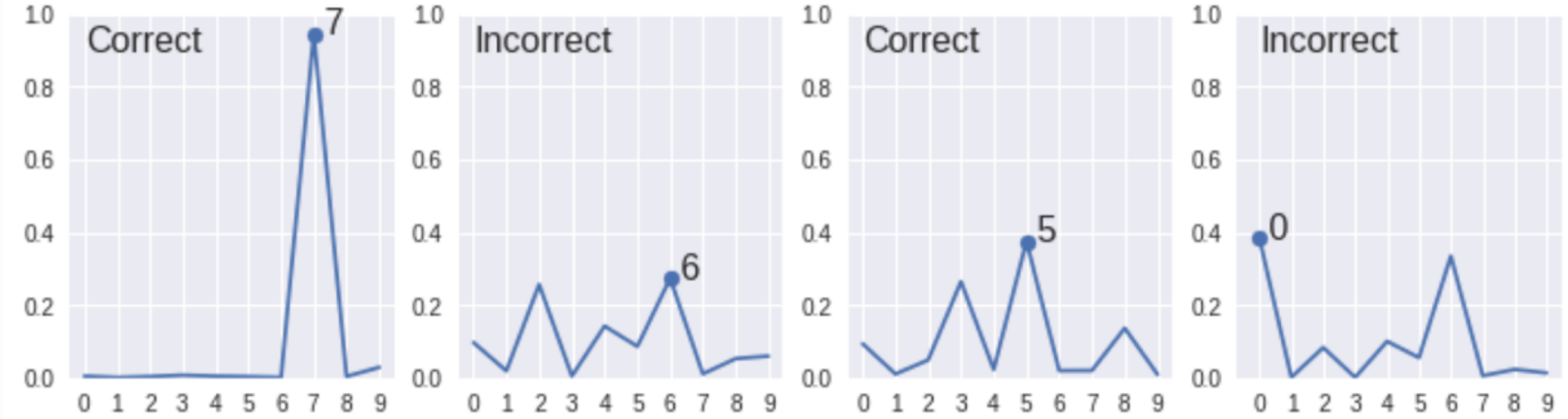

7
5
5
4
After 1 epochs:
Correct
7
Incorrect
2
Incorrect
3
Incorrect
0

7 5 5 4

After 1 epochs:

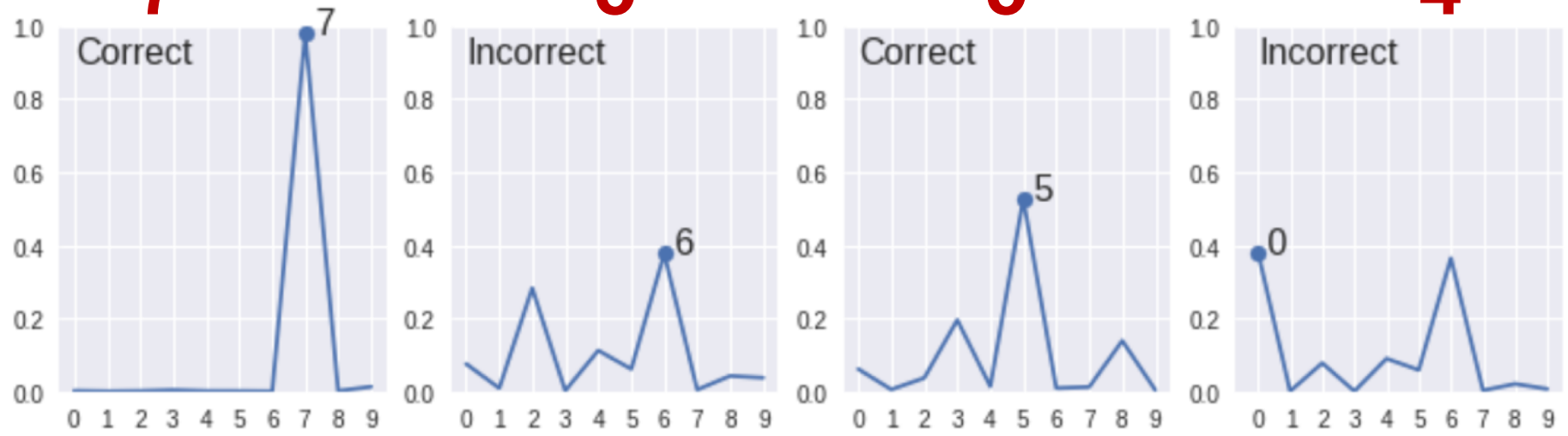


After 2 epochs:

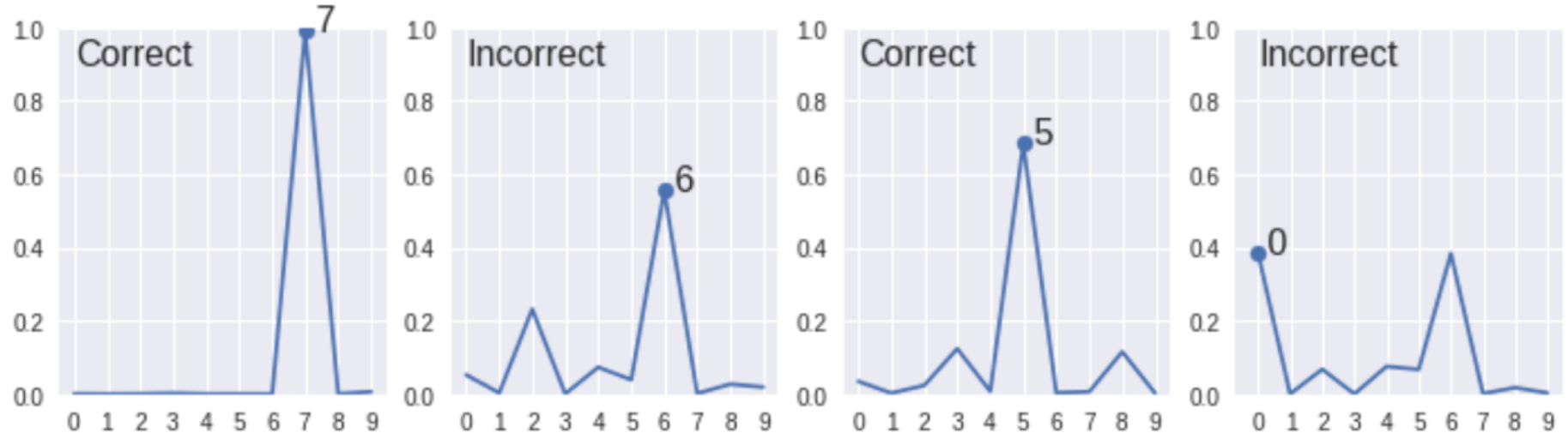


7 5 5 4

After 4 epochs:

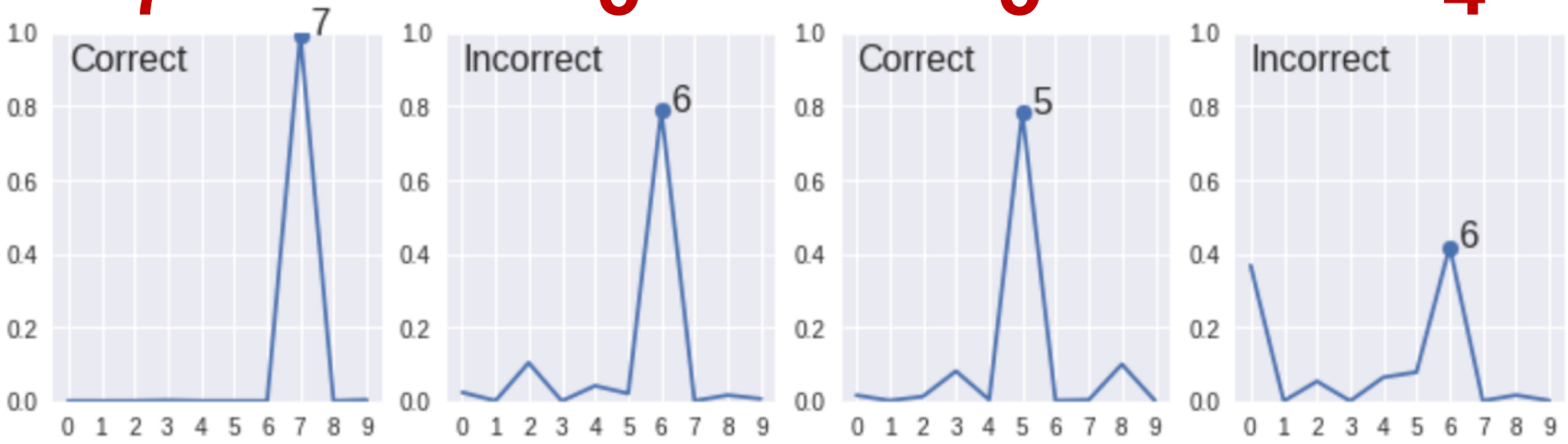


After 8 epochs:

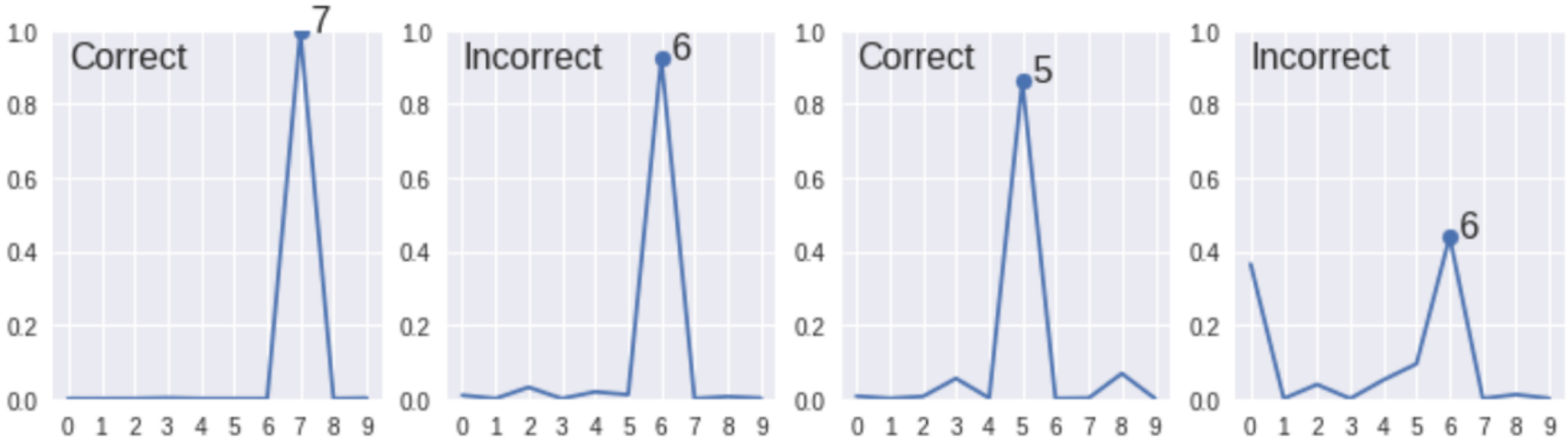


7 5 5 4

After 16 epochs:



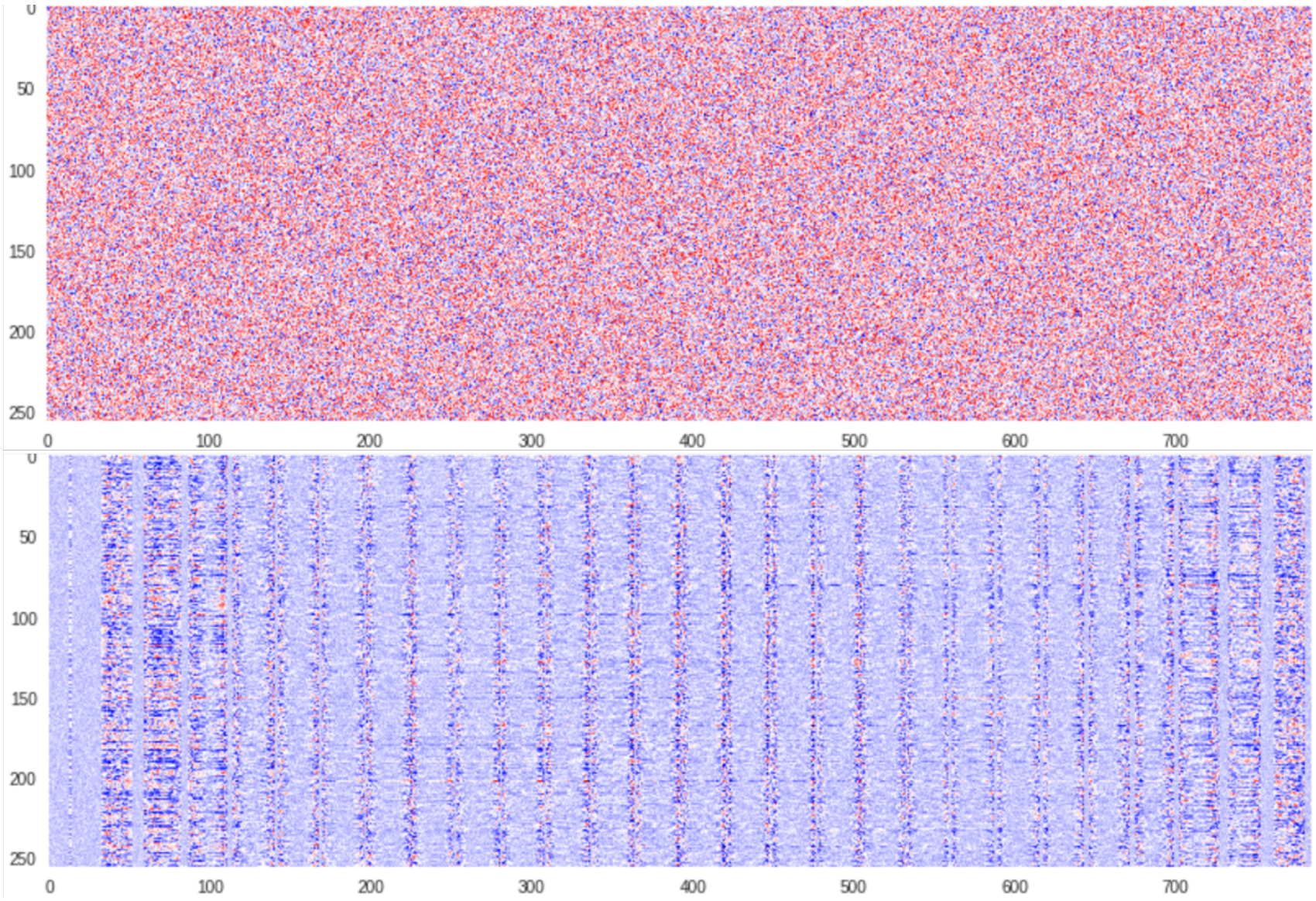
After 32 epochs:



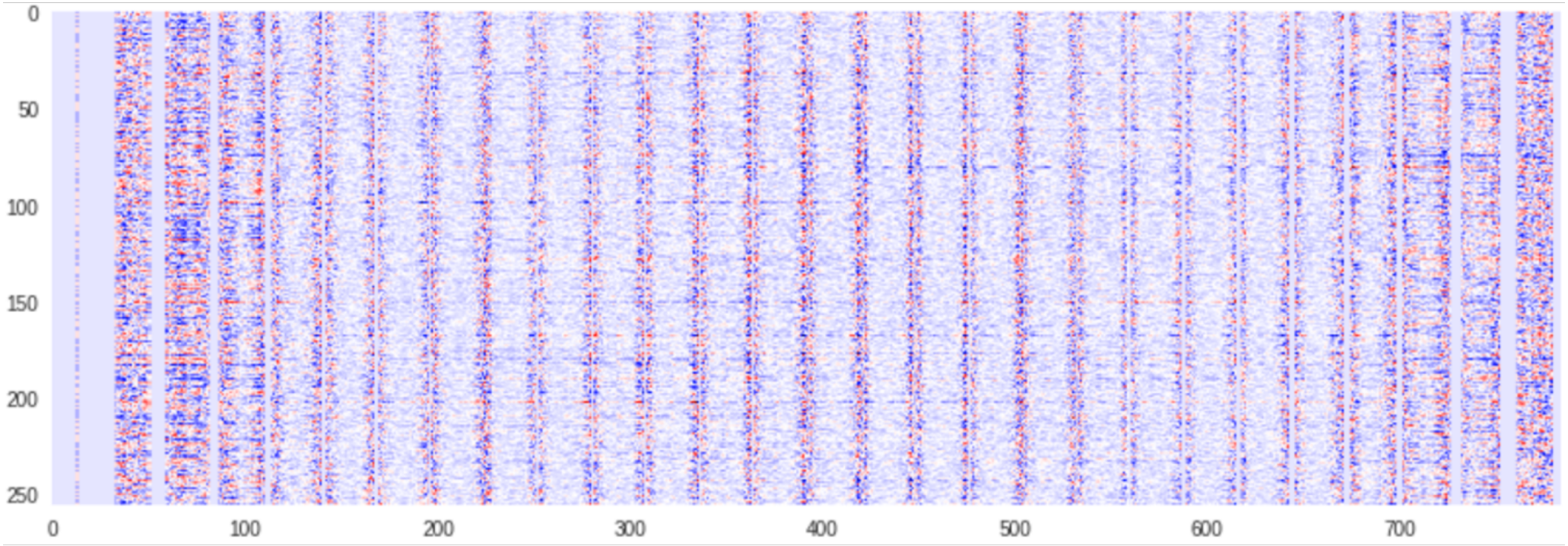
As initialized

Weights, layer 1: 784 x 256

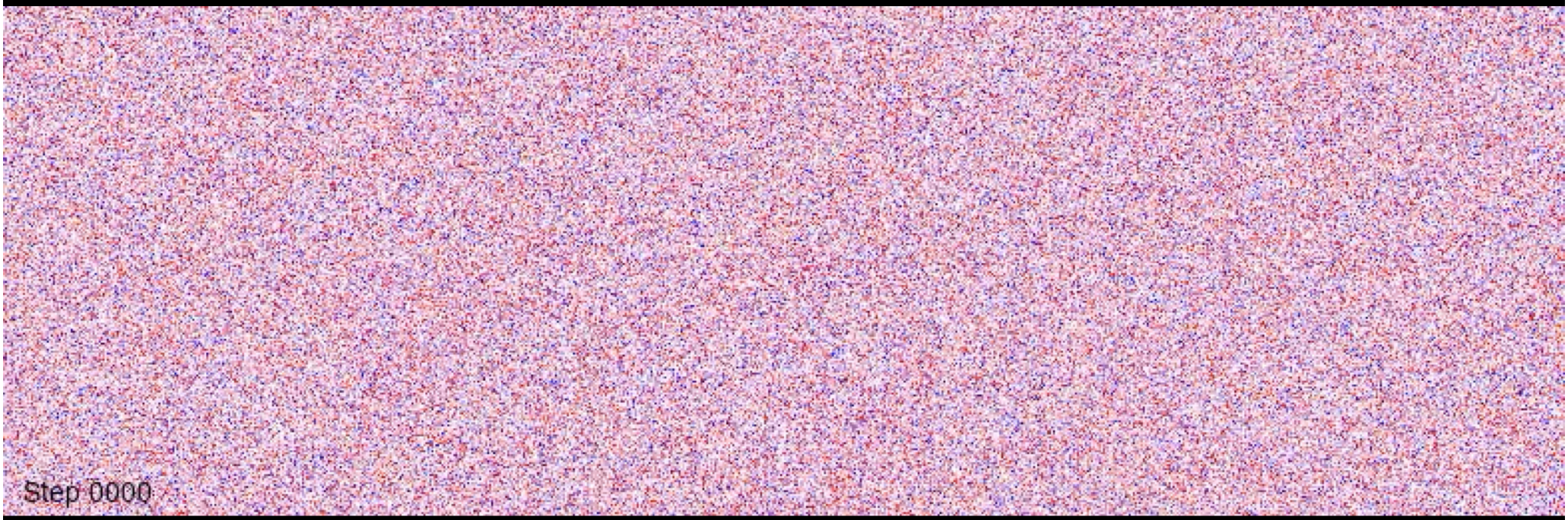
After 5,000 steps



Difference: Layer 1 weights: (after 5000 steps) – (as initialized)



Evolution of Layer 1 Weights, steps 0 to 9999

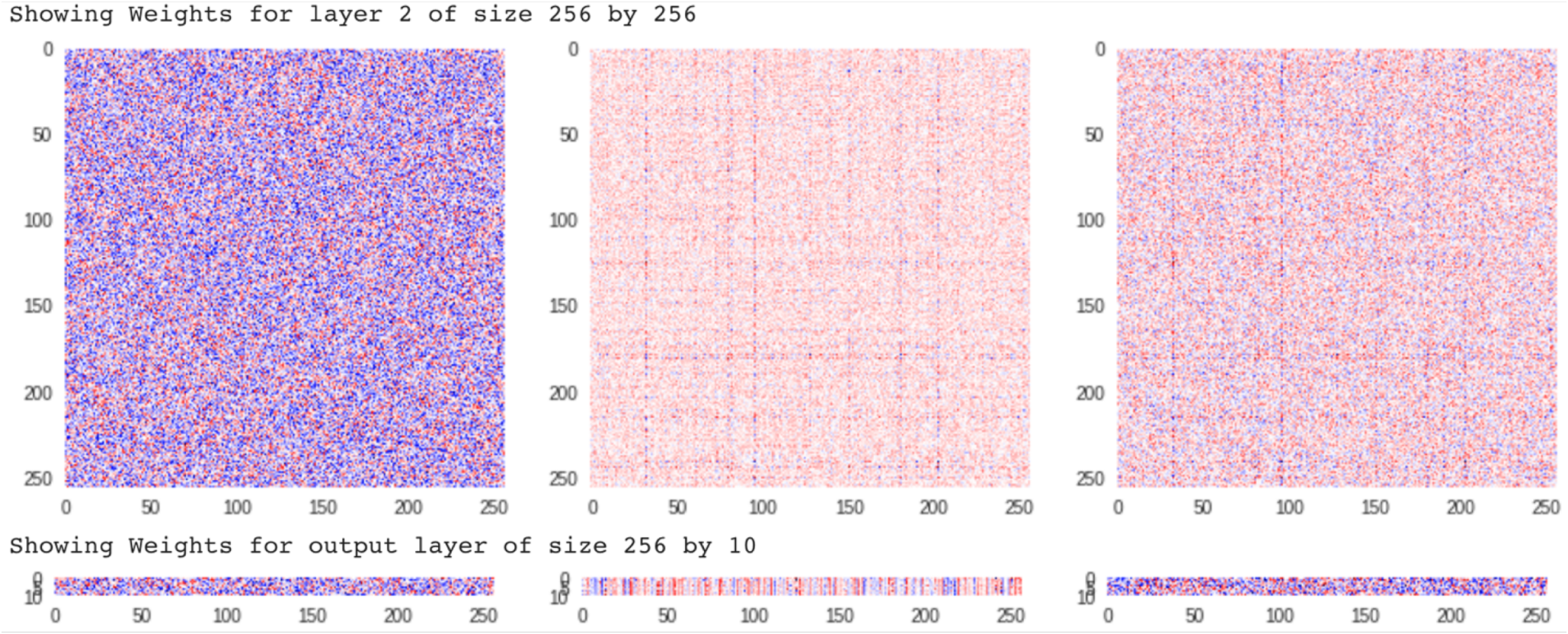


Layer 2 and Output Layer weights

Initial

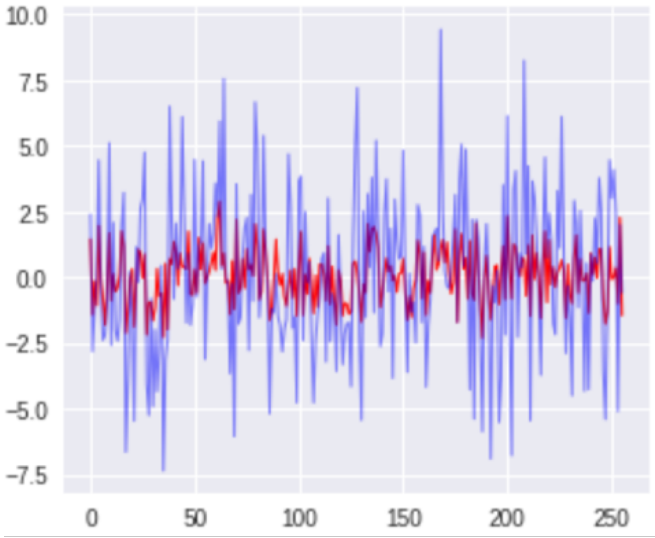
5000 steps

Difference

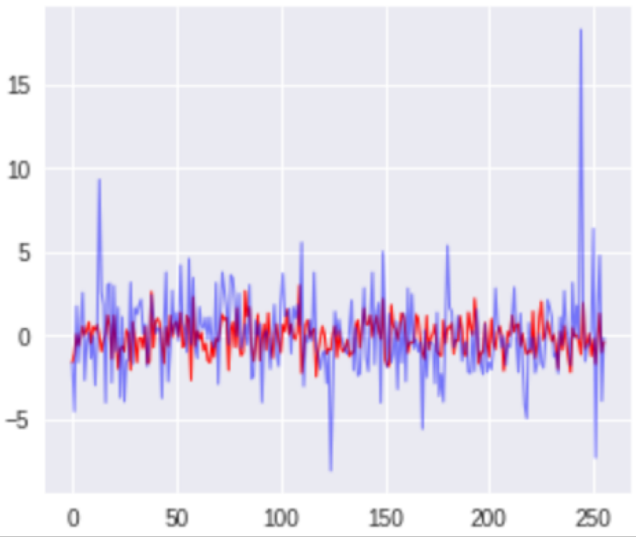


Biases: Initial and after 5000 steps

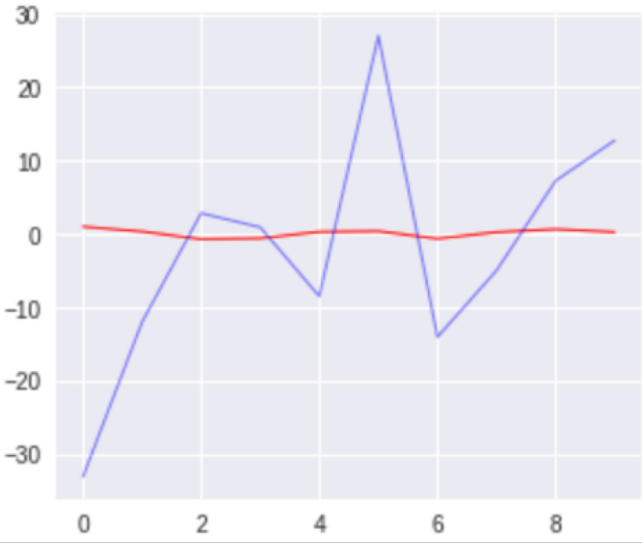
Layer 1



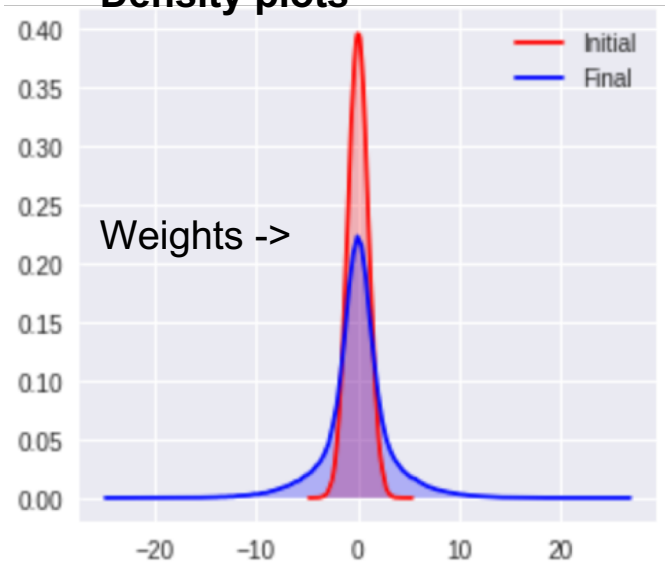
Layer 2



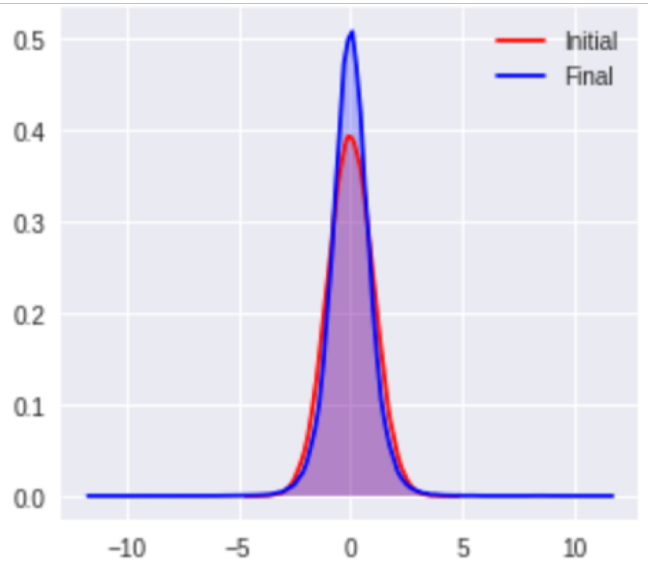
Output

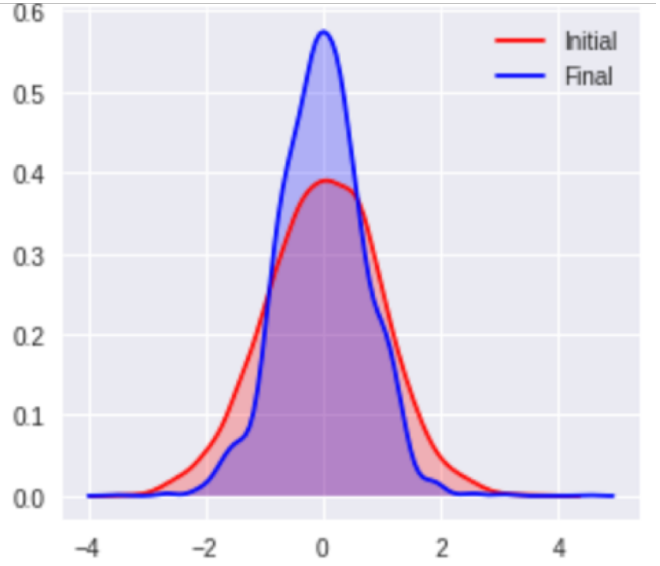


Density plots

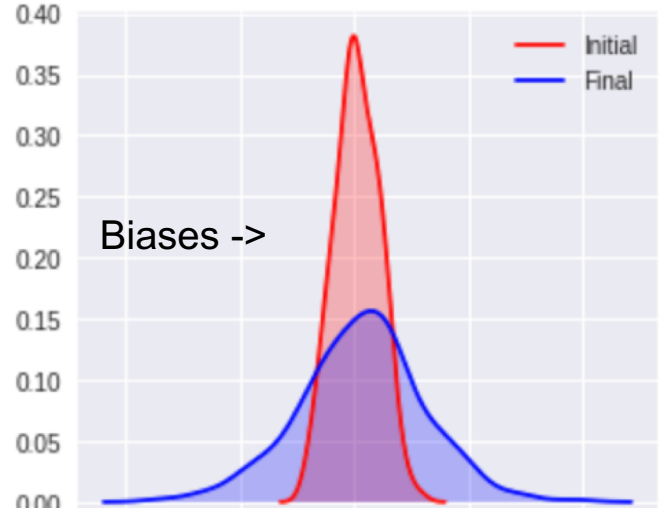


Weights ->

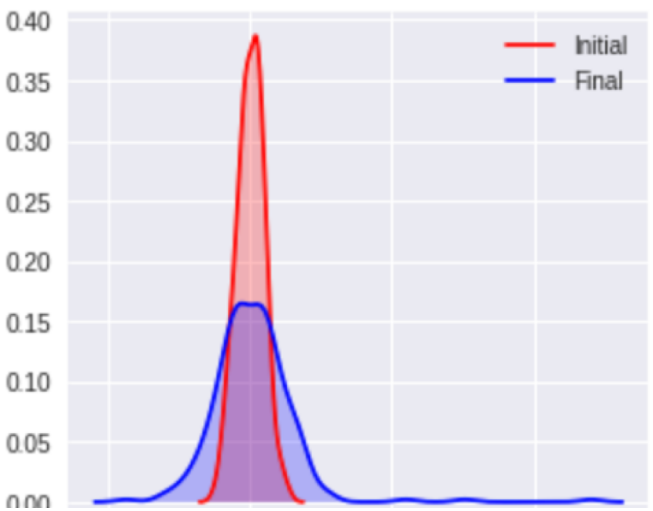


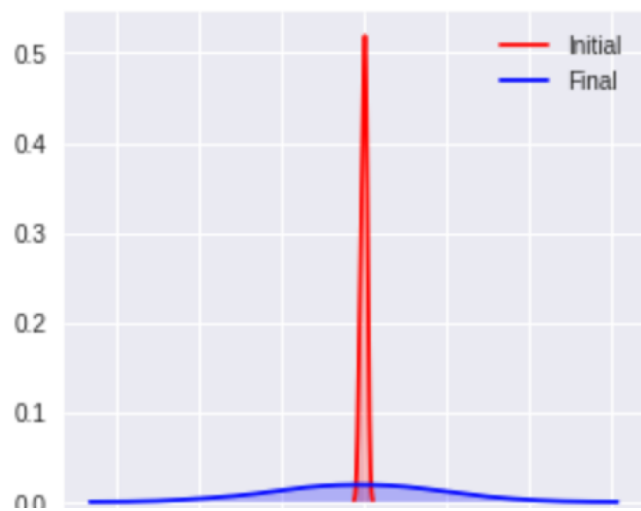


Layer 1 | Layer 2 | Output

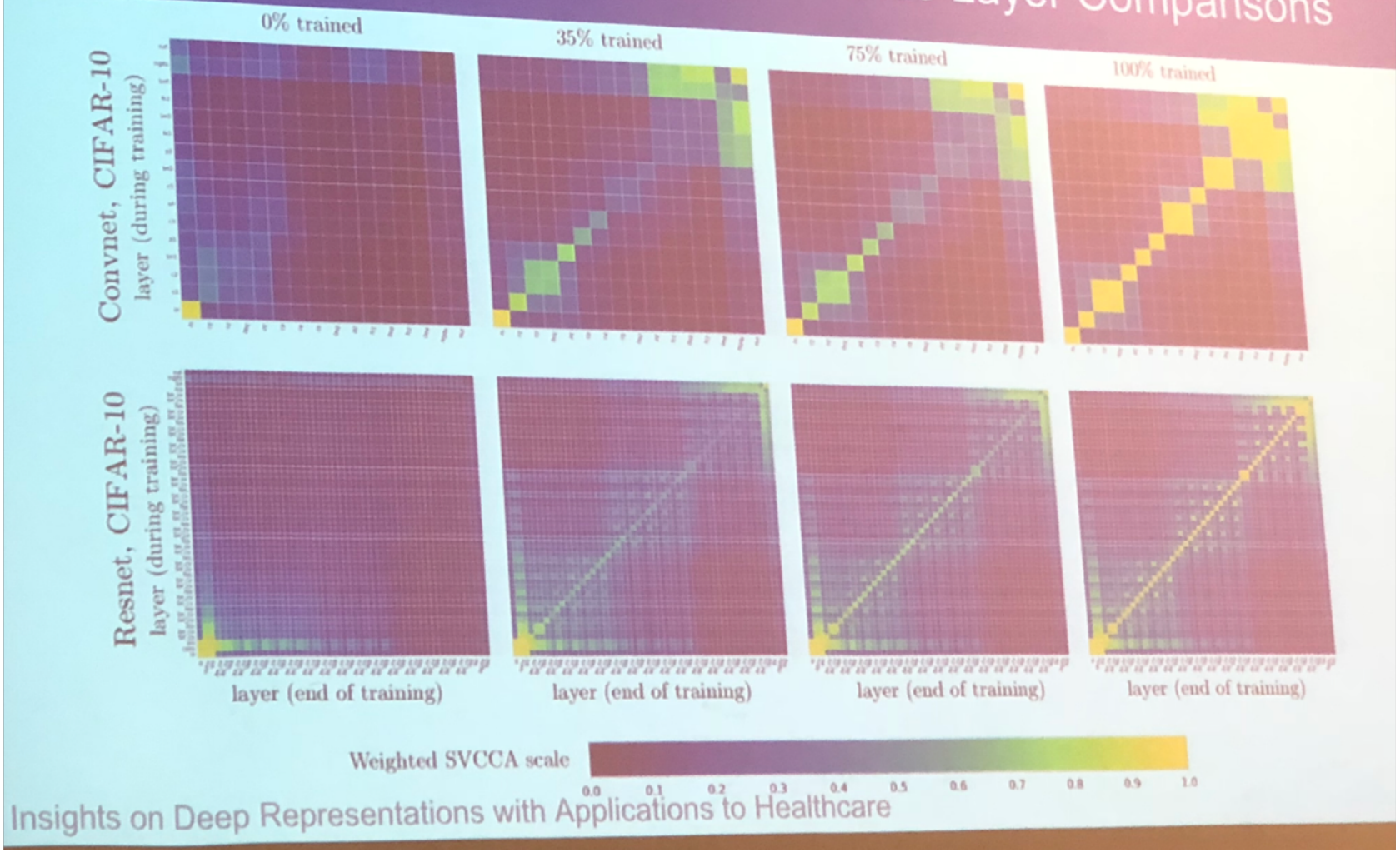


Biases ->





Learning Dynamics: Heatmap of Pairwise Layer Comparisons



Insights on Deep Representations with Applications to Healthcare

How do the weights change?

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))
```

How do the weights change?

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
```

← Here

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))
```

Adjusting weights to reduce loss

Consider our simple neural network:

$$\left.\begin{array}{l} \text{layer\_1} = W_1 * X + B_1 \\ \text{layer\_2} = W_2 * \text{layer\_1} + B_2 \\ \text{out'} = W_3 * \text{layer\_2} + B_3 \\ \text{out} = \text{softmax(out')} \end{array}\right) \quad \textbf{out} = \text{softmax}\left[W_{out}\left(W_2\left(W_1\,\mathbf{x} + B_1\right) + B_2\right) + B_3\right]$$

At a very high level, we train this network as follows:

For each of many steps *t*:

1) for each $(\mathbf{x_i}, \mathbf{y_i})$ in (X, y):

compute $\mathbf{o_i} = \text{softmax}\left[W_{out}\left(W_2\left(W_1\,\mathbf{x_i} + B_1\right) + B_2\right) + B_3\right]$

2) compute loss $L_t = \Sigma_i \left| \mathbf{y_i} - \mathbf{o_i} \right|$

3) adjust W_1, W_2, W_1, B_1, B_2, and B_3 so that, ideally, $L_{t+1} < L_t$ ← How?

For each of many steps *t*:

1) for each ($\mathbf{x_i}$, $\mathbf{y_i}$) in (X, y): compute $\mathbf{o_i} = \text{softmax}[W_{out} (W_2 (W_1 \mathbf{x_i} + B_1) + B_2) + B_3]$
2) compute loss, $L_t = H(\mathbf{y}, \mathbf{o})$ via softmax cross entropy

$$p_i = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad \text{for } i = 1, \ldots, J \qquad H(y,p) = -\sum_i y_i log(p_i)$$

3) adjust W_1, W_2, W_1, B_1, B_2, and B_3 so that, ideally, $L_{t+1} < L_t$ ← How?

Let W = (W_1, W_2, W_3, B_1, B_2, B_3). How do we find W' that minimizes loss?

A) Random: Guess new W'. Accept if L(W') < L(W).

B) Random local search: Try W+δ. Accept if L(W+δ) < L(W).

C) Gradient descent: Follow the gradient of the loss function, *dL/dW*:

```
weights_grad = evaluate_gradient(loss_function, data, weights)
weights += - step_size * weights_grad
```

$$h_i = f(u_i) = f(\sum_{k=1}^{K} w_{ki} x_k)$$

$$y_j = f(u'_j) = f(\sum_{i=1}^{N} w'_{ij} h_i)$$

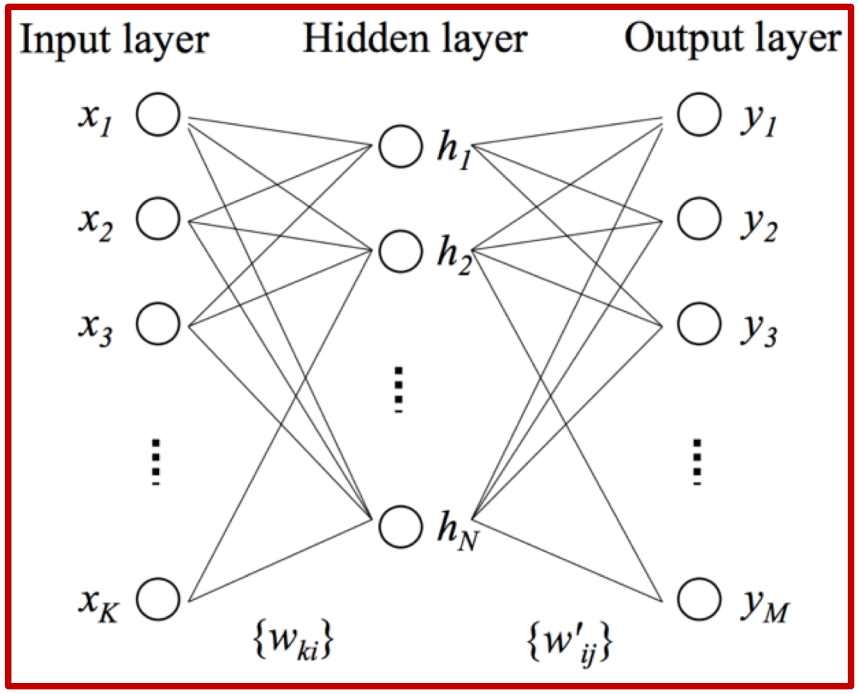
Where:

$$f(u) = \frac{1}{1 + e^u}$$

(sigmoid function)

$$\frac{\partial f}{\partial u} = \mathrm{f(u)(1 - f(u))}$$

$$E = \frac{1}{2} \sum_{j=1}^{M} (y_j - t_j)^2$$



http://alexminnaar.com/deep-learning-basics-neural-networks-backpropagation-and-stochastic-gradient-descent.html

$$h_i = f(u_i) = f(\sum_{k=1}^{K} w_{ki} x_k)$$

$$y_j = f(u'_j) = f(\sum_{i=1}^{N} w'_{ij} h_i)$$

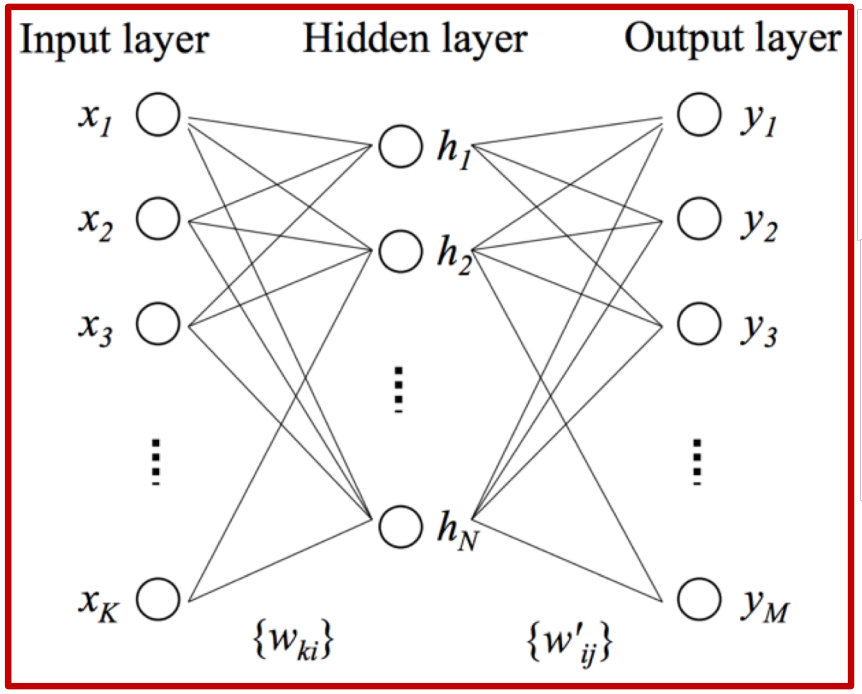
Where:

$$f(u) = \frac{1}{1 + e^u}$$

(sigmoid function)

$$\frac{\partial f}{\partial u} = \mathrm{f(u)(1 - f(u))}$$

$$E = \frac{1}{2} \sum_{j=1}^{M} (y_j - t_j)^2$$



Chain rule:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}}$$

$$\frac{\partial E}{\partial y_j} = y_j - t_j$$

$$\frac{\partial y_j}{\partial u'_j} = y_j(1 - y_j)$$

$$\frac{\partial u'_j}{\partial w'_{ij}} = h_i$$

http://alexminnaar.com/deep-learning-basics-neural-networks-backpropagation-and-stochastic-gradient-descent.html

$$h_i = f(u_i) = f(\sum_{k=1}^{K} w_{ki} x_k)$$

$$y_j = f(u'_j) = f(\sum_{i=1}^{N} w'_{ij} h_i)$$

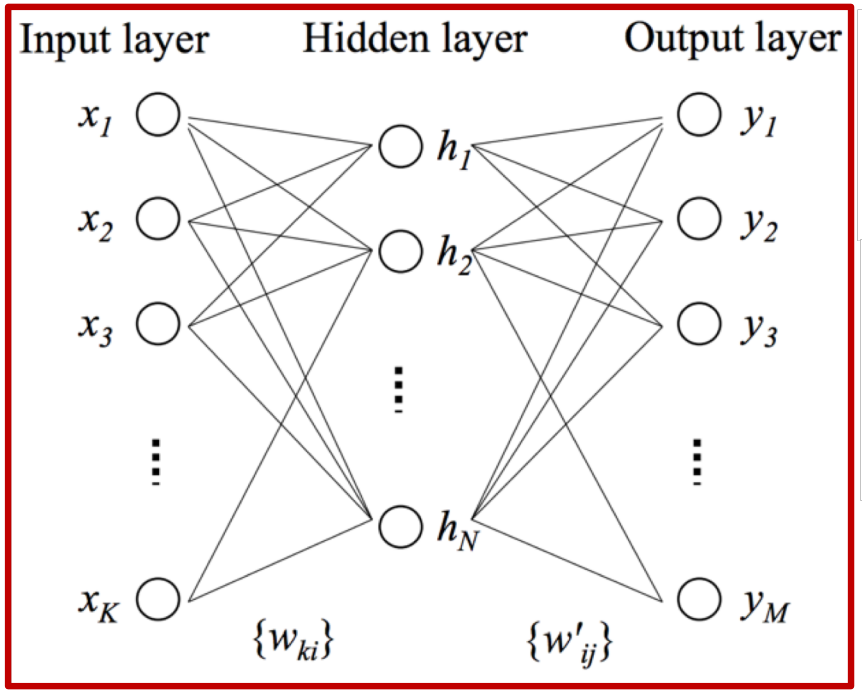
Where:

$$f(u) = \frac{1}{1 + e^u}$$

(sigmoid function)

$$\frac{\partial f}{\partial u} = \mathrm{f(u)(1 - f(u))}$$

$$E = \frac{1}{2} \sum_{j=1}^{M} (y_j - t_j)^2$$



Chain rule:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial w'_{ij}}$$

$$\frac{\partial E}{\partial y_j} = y_j - t_j$$

$$\frac{\partial y_j}{\partial u'_j} = y_j(1 - y_j)$$

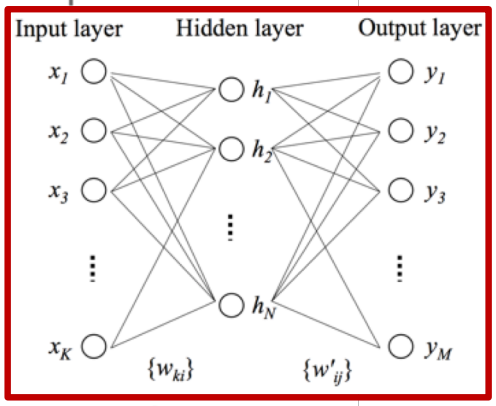
$$\frac{\partial u'_j}{\partial w'_{ij}} = h_i$$

$$w'^{new}_{ij} = w'^{old}_{ij} - \eta \cdot (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i$$

$$\frac{\partial E}{\partial w'_{ij}} = (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i$$

http://alexminnaar.com/deep-learning-basics-neural-networks-backpropagation-and-stochastic-gradient-descent.html

Now let's turn our attention to the gradient of the objective function with respect to the input-to-hidden weights w_{ki}. As we shall see, this gradient has already been partially computed when we computed the previous gradient.



Using the chain rule, the full gradient is

$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{M} \left(\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} \cdot \frac{\partial u'_j}{\partial h_i} \right) \cdot \frac{\partial h_i}{\partial u_i} \cdot \frac{\partial u_i}{\partial w_{ki}}$$

The sum is due to the fact that the hidden unit that w_{ki} connects to is itself connected to every output unit, thus each of these gradients need to be taken into account as well. We have already computed both $\frac{\partial E}{\partial y_j}$ and $\frac{\partial y_j}{\partial u'_j}$ which means that

$$\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u'_j} = (y_j - t_j) \cdot y_j(1 - y_j)$$

Now we need to compute the remaining derivatives $\frac{\partial u'_j}{\partial h_i}$, $\frac{\partial h_i}{\partial u_i}$, and $\frac{\partial u_i}{\partial w_{ki}}$. So let's do just that.

$$\frac{\partial u'_j}{\partial h_i} = \frac{\partial \sum_{i=1}^{N} w'_{ij} h_i}{\partial h_i} = w'_{ij}$$

and, again using the derivative of the logistic function

(sigmoid)

$$\frac{\partial h_i}{\partial u_i} = h_i(1 - h_i)$$

and finally

$$\frac{\partial u_i}{\partial w_{ki}} = \frac{\partial \sum_{k=1}^{K} w_{ki} x_k}{\partial w_{ki}} = x_k$$

After making the appropriate substitutions we arrive at the gradient

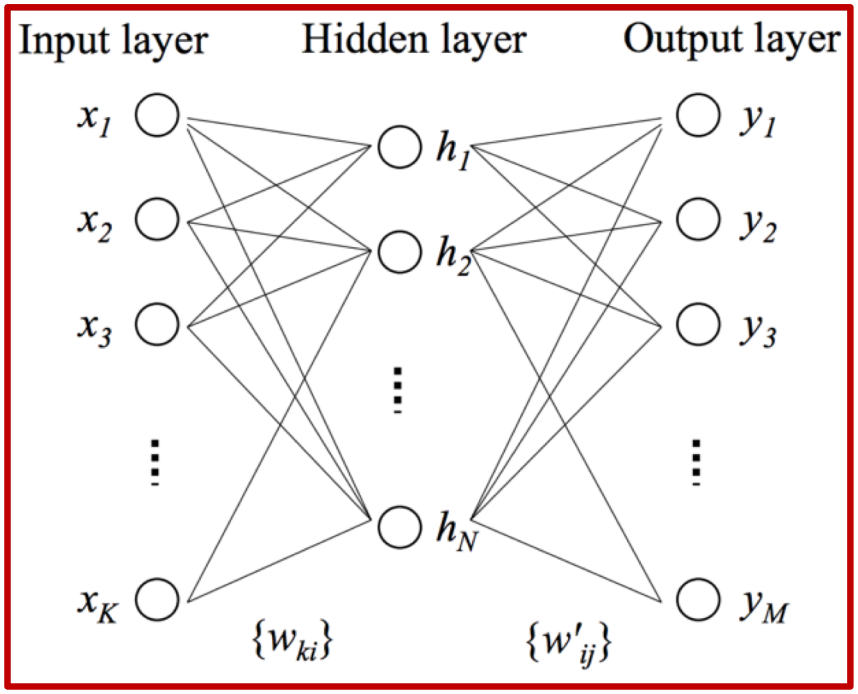
$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{M} [(y_j - t_j) \cdot y_j(1 - y_j) \cdot w'_{ij}] \cdot h_i(1 - h_i) \cdot x_k$$

And the update equation becomes

$$w_{ki}^{new} = w_{ki}^{old} - \eta \cdot \sum_{j=1}^{M} [(y_j - t_j) \cdot y_j(1 - y_j) \cdot w'_{ij}] \cdot h_i(1 - h_i) \cdot x_k$$

Back propagation

Thus, output error at node j, $y_j - t_j$, is propagated back through network:



$$w'^{new}_{ij} = w'^{old}_{ij} - \eta \cdot (y_j - t_j) \cdot y_j(1 - y_j) \cdot h_i$$

$$w^{new}_{ki} = w^{old}_{ki} - \eta \cdot \sum_{j=1}^{M} [(y_j - t_j) \cdot y_j(1 - y_j) \cdot w'_{ij}] \cdot h_i(1 - h_i) \cdot x_k$$

We use gradients to update weights

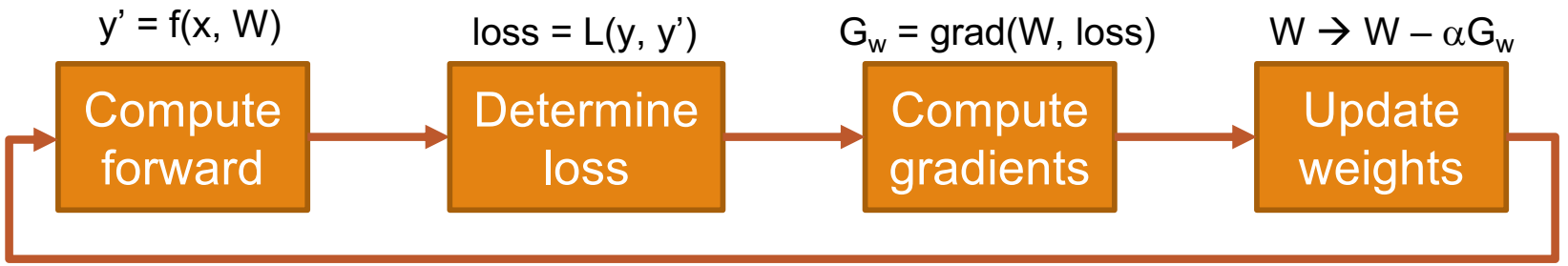
train_step calls **minimize**, which applies gradient updates to the variables. It first **computes gradients** and then **applies gradients**.

Repeat until convergence {

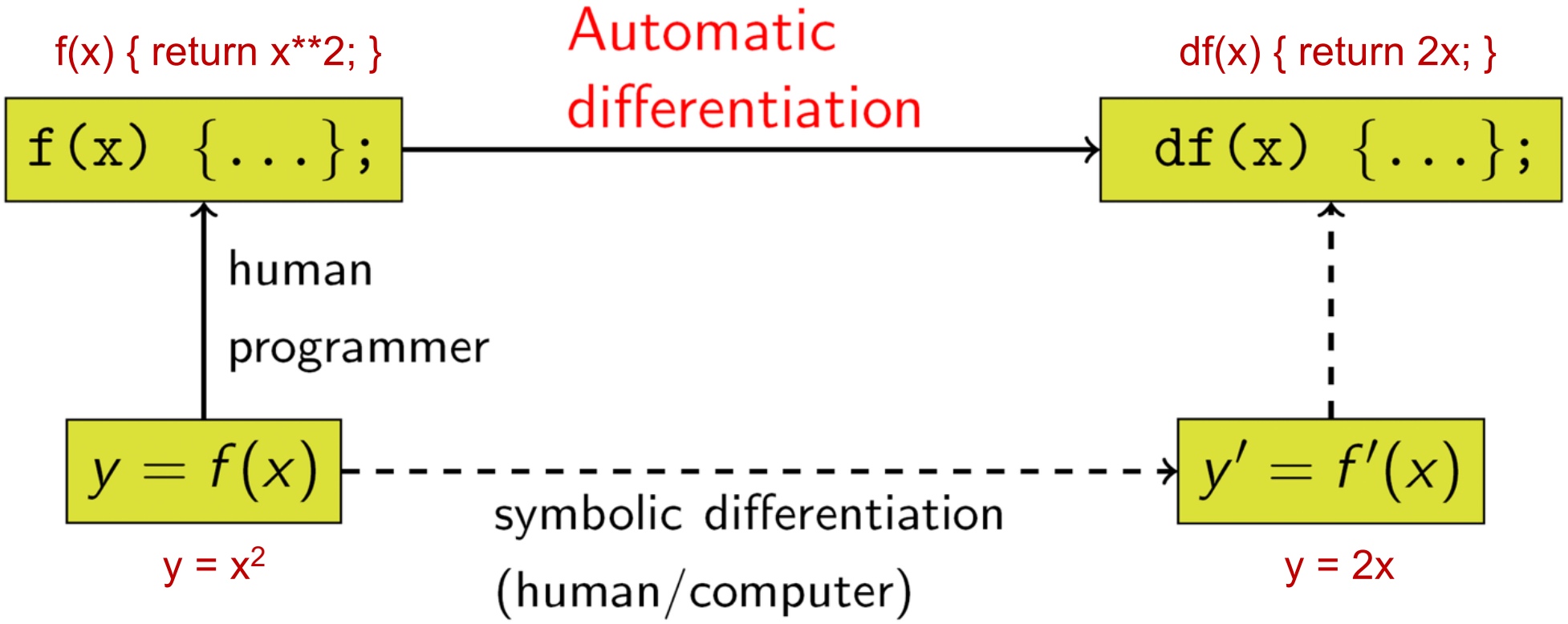
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

```
# Compute the gradient of cost with respect to W and b
grad_W, grad_b = tf.gradients(xs=[W, b], ys=cost)

# Apply gradients to W and b
new_W = W.assign(W - learning_rate * grad_W)
new_b = b.assign(b - learning_rate * grad_b)
```



We find the gradient of the loss function by Automation Differentiation



https://en.wikipedia.org/wiki/Automatic_differentiation

How TensorFlow uses AD

Fundamental to AD is the decomposition of differentials provided by the **chain rule**. For the simple composition:

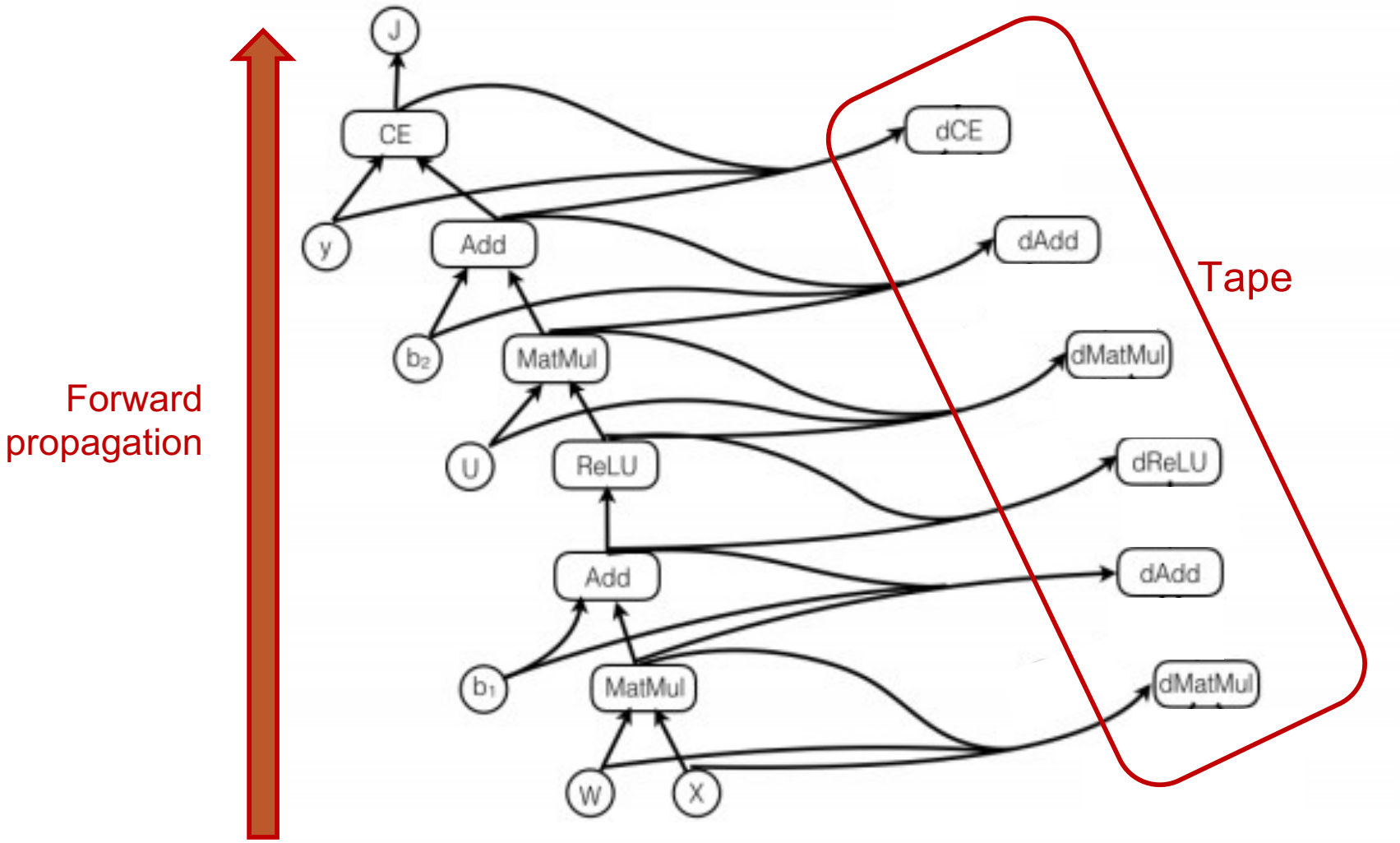
$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3 \text{ the chain rule gives}$$

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

TensorFlow first "records" all the operations applied to compute the output of the function. We call this record a "**tape.**" It then uses that tape and the gradients functions associated with each primitive operation to compute the gradients of the user-defined function using **reverse mode differentiation**.

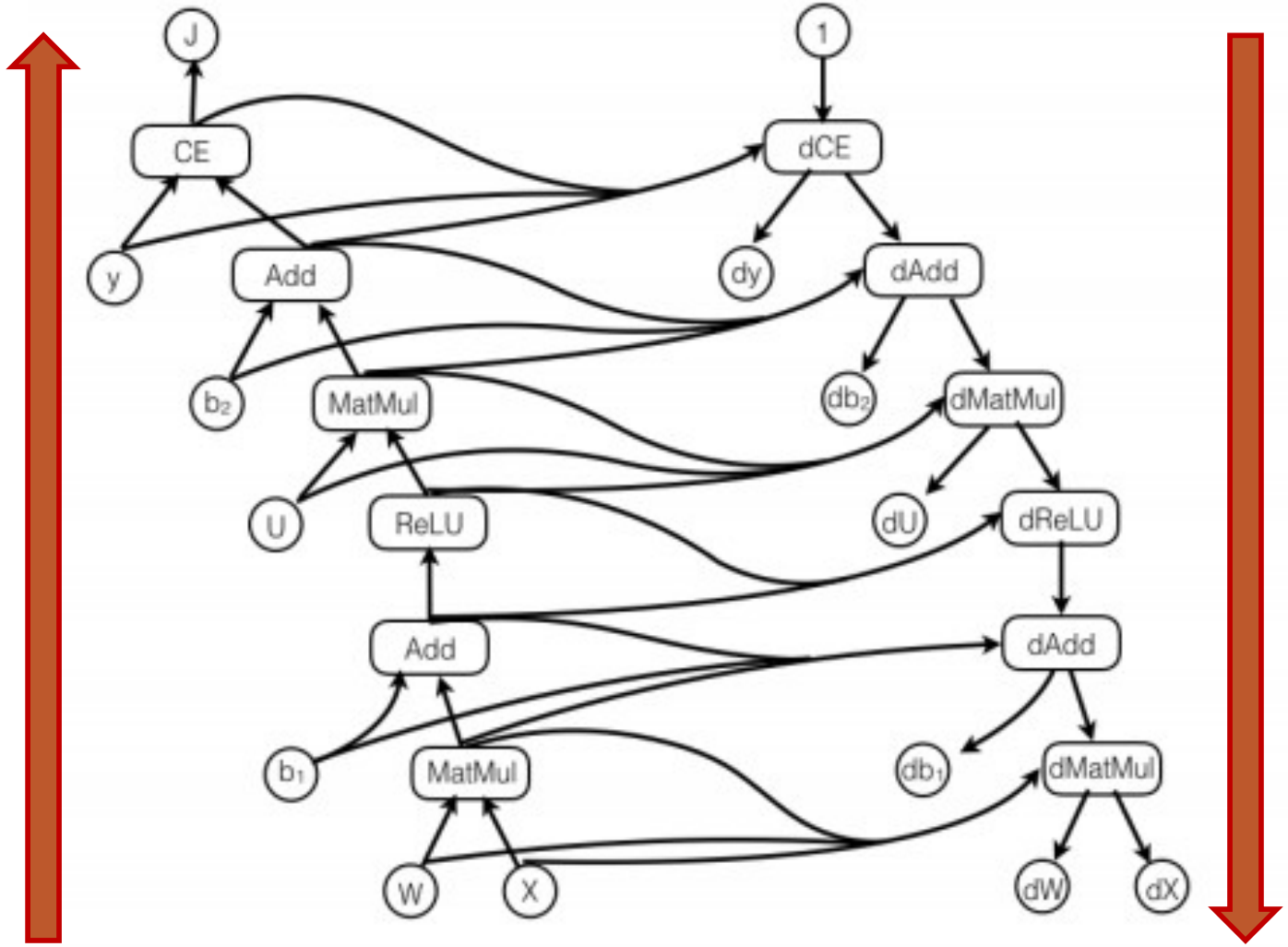
reverse accumulation computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$.

https://www.tensorflow.org/tutorials/eager/automatic_differentiation



https://www.linkedin.com/pulse/how-tensorflow-calculate-gradients-shamane-siriwardhana/

Forward propagation



Backward propagation

https://www.linkedin.com/pulse/how-tensorflow-calculate-gradients-shamane-siriwardhana/

Gradient descent

How many (xi, yi) pairs do we evaluate before updating the weights?

- **All** = Gradient descent
- **1** = Stochastic gradient descent
- **N** = Mini-batch gradient descent

Gradient of loss = mean of gradient over each sample loss

$$\nabla L = \frac{1}{n} \sum_{x} \nabla L_x$$

```
while True:
    data_batch = sample_training_data(data, 256)   # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad    # perform parameter update
```

```
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

MNIST in Tensorflow

It’s linear algebra all the way down

Observations

- It's all linear algebra, with matrix sizes depending on network and data
- There are surely tradeoffs between numerical resolution, speed, and accuracy
- There is a lot of data reading