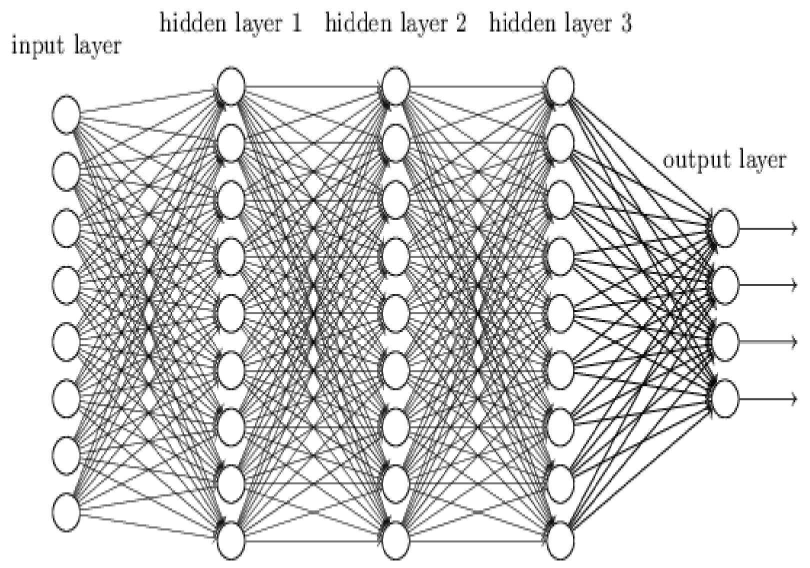
Deep Learning and Convolutional Neural Networks

Instructor: Dr. Mohammad Rashedur Rahman

Fully Connected Back Propagation Neural Network

- We learned networks in which adjacent network layers are fully connected to one another. That is, every neuron in the network is connected to every neuron in adjacent layers
- For the 28×28 pixel images we've been using, this means our network has 784 (=28×28) input neurons. We then trained the network's weights and biases so that the network's output would - we hope! - correctly identify the input image: '0', '1', '2', ..., '8', or '9'.
- For each pixel in the input image, we encoded the pixel's intensity as the value for a corresponding neuron in the input layer.
- Our earlier networks work pretty well: we've obtained a classification accuracy better than 98 percent, using training and test data from the MNIST handwritten digit data set.



Good, but some stories not yet revealed

- Such a network architecture does not take into account the spatial structure of the images.
- For instance, it treats input pixels which are far apart and close together on exactly the same footing.
- Such concepts of spatial structure must instead be inferred from the training data. But what if, instead of starting with a network architecture, we used an architecture which tries to take advantage of the spatial structure?
- Those are basically *convolutional neural networks (CNN).*
- Using this architecture makes convolutional networks fast to train.
- This, in turn, helps us train deep, many-layer networks, which are very good at classifying images. Today, deep convolutional networks or some close variant are used in most neural networks for image recognition

Basic Ideas of CNN

Convolutional neural networks use three basic ideas:

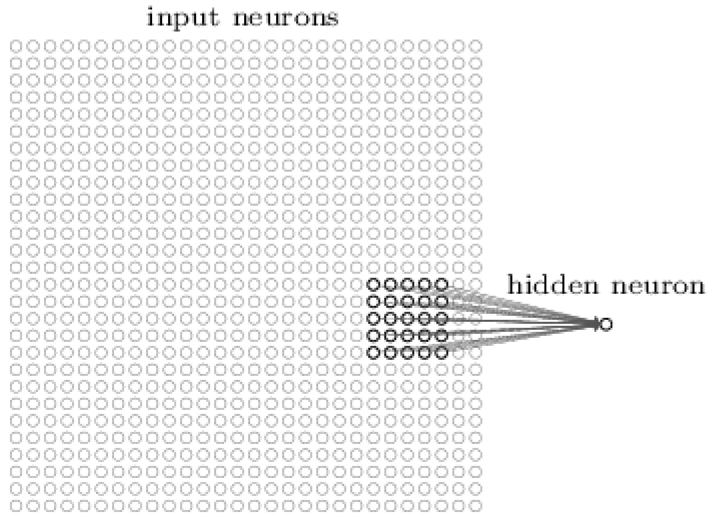
1. local receptive fields,

2. shared weights, and

3. pooling.

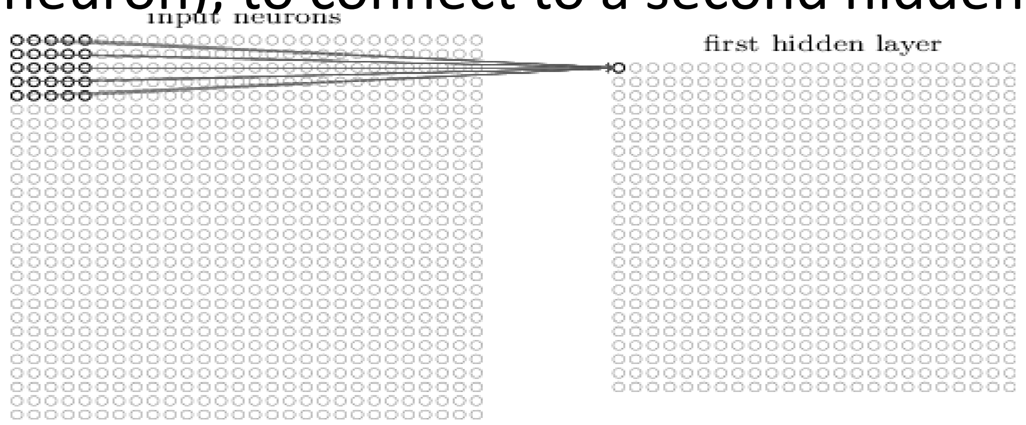
CNN: Local Receptive Field

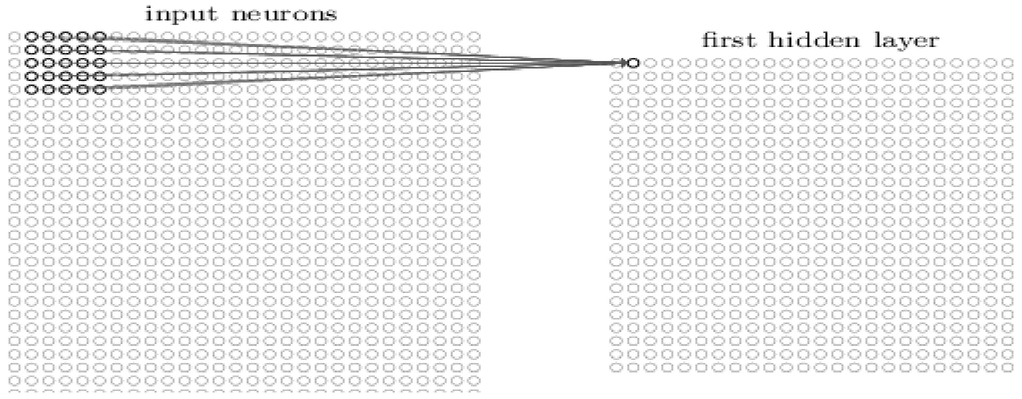
- As per usual, we'll connect the input pixels to a layer of hidden neurons. But we won't connect every input pixel to every hidden neuron. Instead, we only make connections in small, localized regions of the input image.
- To be more precise, each neuron in the first hidden layer will be connected to a small region of the input neurons, say, for example, a 5X5 region corresponding to 25 pixels, So, for a particular hidden neuron, we might have connections that look like this:



CNN: Local Receptive Field (Cont..)

- That region in the input image is called the *local receptive field* for the hidden neuron.
- It's a little window on the input pixels. Each connection learns a weight. And the hidden neuron learns an overall bias as well. You can think of that particular hidden neuron as learning to analyze its particular local receptive field.
- We then slide the local receptive field across the entire input image. For each local receptive field, there is a different hidden neuron in the first hidden layer. To illustrate this concretely, let's start with a local receptive field in the top-left corner:
- Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:





CNN: Local Receptive Field (Cont..)

- And so on, building up the first hidden layer. Note that if we have a 28×28 input image, and 5×5 local receptive fields, then there will be 24×24 neurons in the hidden layer.
- This is because we can only move the local receptive field 23 neurons across (or 23 neurons down), before colliding with the right-hand side (or bottom) of the input image.
- We have shown the local receptive field being moved by one pixel at a time. In fact, sometimes a different ***stride length*** is used.
- For instance, we might move the local receptive field 22 pixels to the right (or down), in which case we'd say a stride length of 22 is used.

Shared Weights

- I've said that each hidden neuron has a bias and 5×5 weights connected to its local receptive field.
- What I did not yet mention is that we're going to use the *same weights* and *bias* for each of the 24×24 hidden neurons. In other words, for the j,k th hidden neuron, the output is:

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right).$$

- Here, σ is the neural activation function - perhaps the sigmoid function we used in earlier chapters. b is the shared value for the bias. $W_{l,m}$ is a 5×5 array of shared weights. And, finally, we use $a_{x,y}$ to denote the input activation at position x,y.

Shared Weights (Cont..)

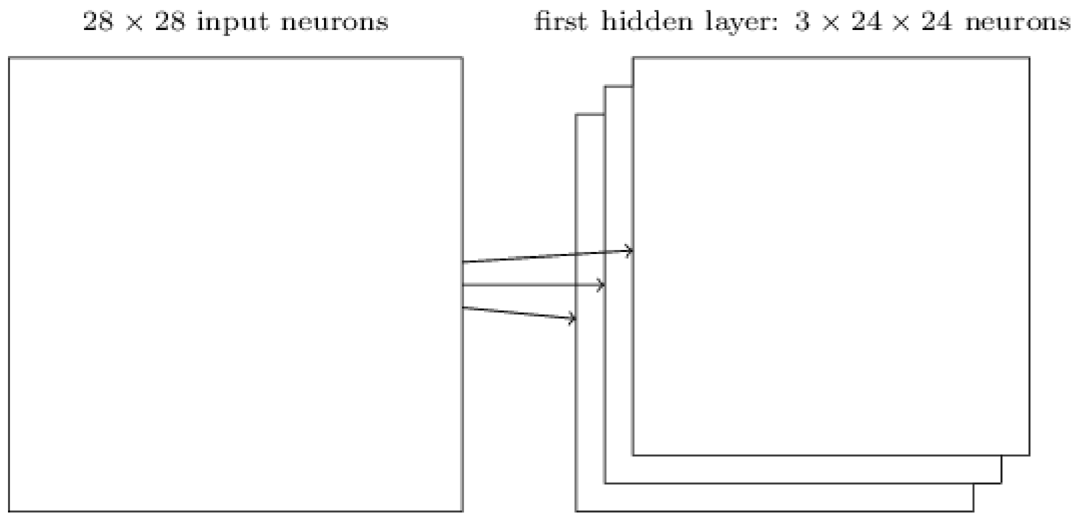
- This means that all the neurons in the first hidden layer detect exactly the same feature. Informally, think of the feature detected by a hidden neuron as the kind of input pattern that will cause the neuron to activate.
- it might be an edge in the image, for instance, or maybe some other type of shape., just at different locations in the input image.
- To see why this makes sense, suppose the weights and bias are such that the hidden neuron can pick out, say, a vertical edge in a particular local receptive field.
- That ability is also likely to be useful at other places in the image. And so it is useful to apply the same feature detector everywhere in the image.
- To put it in slightly more abstract terms, convolutional networks are well adapted to the translation invariance of images: move a picture of a cat (say) a little ways, and it's still an image of a cat

Shared Weights (Cont..)

- For this reason, we sometimes call the map from the input layer to the hidden layer a *feature map*.
- We call the weights defining the feature map the *shared weights*. And we call the bias defining the feature map in this way the *shared bias*.
- The shared weights and bias are often said to define a *kernel* or *filter*. In the literature, people sometimes use these terms in slightly different ways

Shared Weights (Cont..)

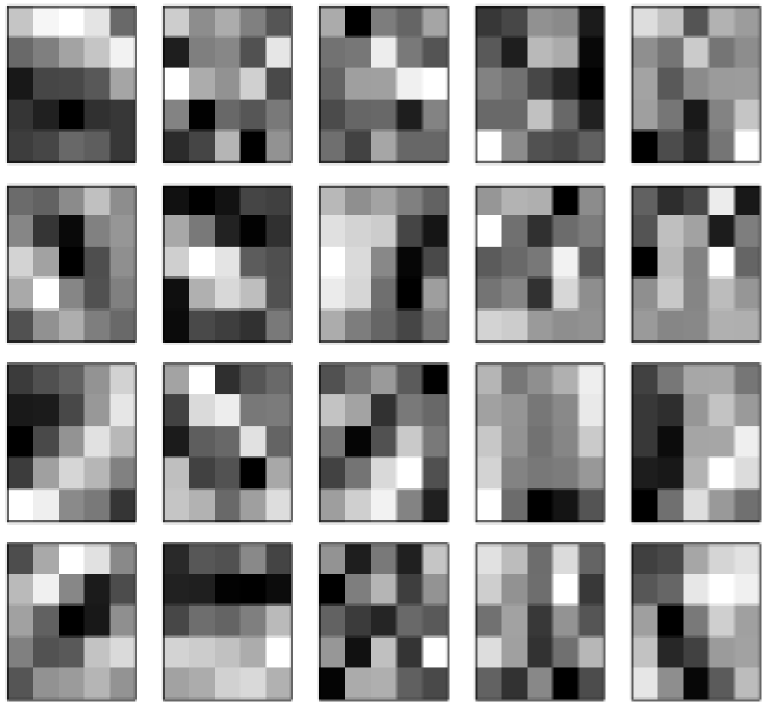
- The network structure I've described so far can detect just a single kind of localized feature.
- To do image recognition we'll need more than one feature map.
- And so a complete convolutional layer consists of several different feature maps:



- However, in practice convolutional networks may use more (and perhaps many more) feature maps. One of the early convolutional networks, LeNet-5, used 66 feature maps, each associated to a 5×5 local receptive field, to recognize MNIST digits

Shared Weights (Cont..)

- The 20 images correspond to 20 different feature maps (or filters, or kernels).
- Each map is represented as a 5×5 block image, corresponding to the 5×5 weights in the local receptive field.
- Whiter blocks mean a smaller (typically, more negative) weight, so the feature map responds less to corresponding input pixels.
- Darker blocks mean a larger weight, so the feature map responds more to the corresponding input pixels. Very roughly speaking, the images above show the type of features the convolutional layer responds to .



Shared Weights (Cont..)

- So what can we conclude from these feature maps?
- It's clear there is spatial structure here beyond what we'd expect at random: many of the features have clear sub-regions of light and dark.
- That shows our network really is learning things related to the spatial structure.
- However, beyond that, it's difficult to see what these feature detectors are learning.

Shared Weights (Cont..)

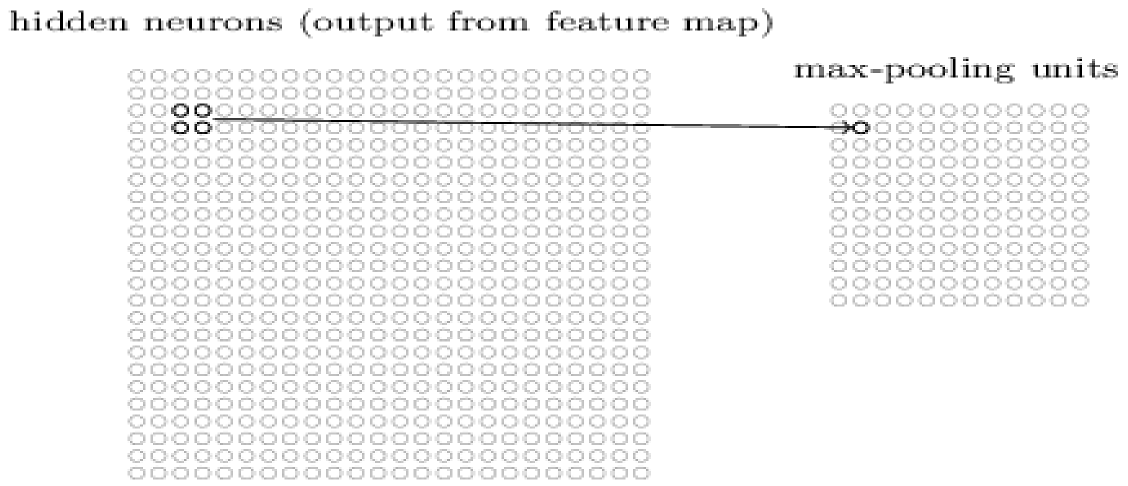
- A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network.
- For each feature map we need 25=5×5 shared weights, plus a single shared bias. So each feature map requires 26 parameters.
- If we have 20 feature maps that's a total of 20×26=520 parameters defining the convolutional layer.
- By comparison, suppose we had a fully connected first layer, with 784=28×28 input neurons, and a relatively modest 30 hidden neurons
- That's a total of 784×30 weights, plus an extra 30 biases, for a total of 23,550 parameters. In other words, the fully-connected layer would have more than 40 times as many parameters as the convolutional layer.

Pooling Layers

- In addition to the convolutional layers just described, convolutional neural networks also contain *pooling layers*.
- Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer.
- A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map.
- For instance, each unit in the pooling layer may summarize a region of (say) 2×2 neurons in the previous layer. As a concrete example, one common procedure for pooling is known as *max-pooling*

Pooling Layers (cont..)

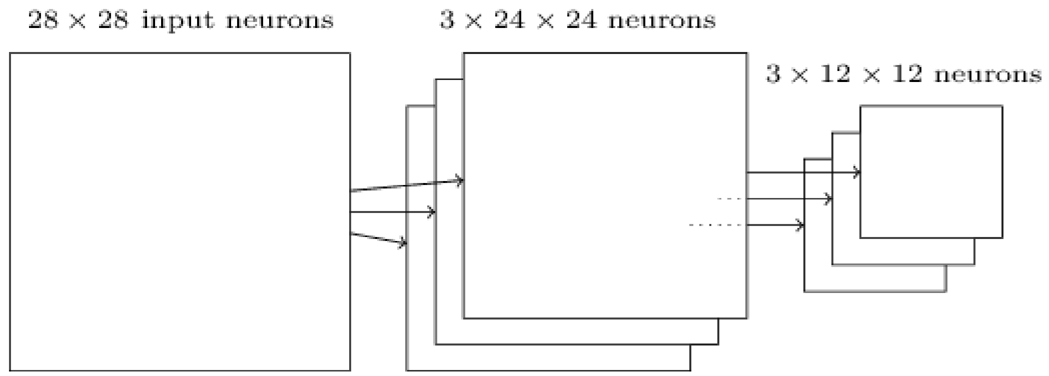
- In max-pooling, a pooling unit simply outputs the maximum activation in the 2×2 input region, as illustrated in the following diagram:



- Note that since we have 24×24 neurons output from the convolutional layer, after pooling we have 12×12 neurons.

Pooling Layers (cont..)

- The convolutional layer usually involves more than a single feature map. We apply max-pooling to each feature map separately. So if there were three feature maps, the combined convolutional and max-pooling layers would look like:



28×28 input neurons
$3 \times 24 \times 24$ neurons
$3 \times 12 \times 12$ neurons

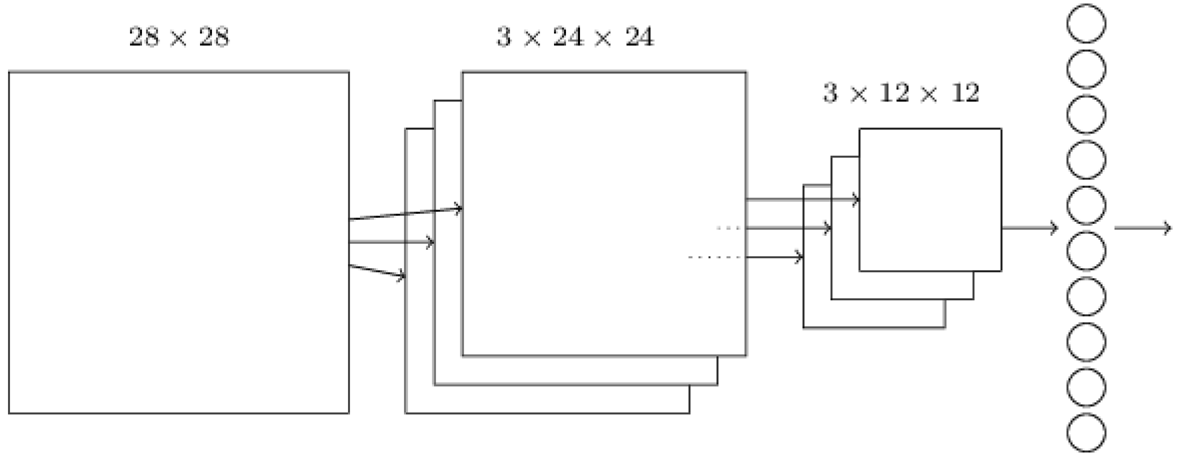
- We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image.
- It then throws away the exact positional information.
- The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features.
- A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers

Pooling Layers (cont..)

- Max-pooling isn't the only technique used for pooling. Another common approach is known as *L2 pooling*.
- Here, instead of taking the maximum activation of a 2×2 region of neurons, we take the square root of the sum of the squares of the activations in the 2×2region.
- While the details are different, the intuition is similar to max-pooling: L2 pooling is a way of condensing information from the convolutional layer.

Putting All Together

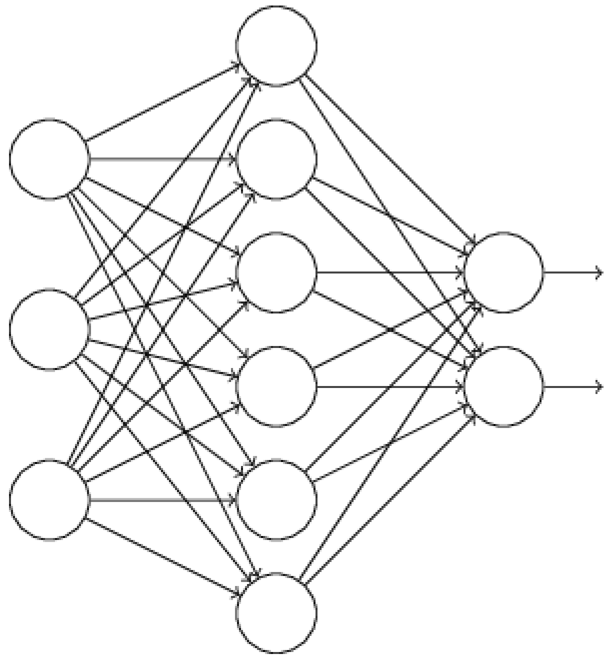
- We can now put all these ideas together to form a complete convolutional neural network.
- It's similar to the architecture we were just looking at, but has the addition of a layer of 10 output neurons, corresponding to the 10 possible values for MNIST digits ('0', '1', '2', *etc*):



- In this architecture, we can think of the convolutional and pooling layers as learning about local spatial structure in the input training image, while the later, fully-connected layer learns at a more abstract level, integrating global information from across the entire image. This is a common pattern in convolutional neural networks.

Dropout

- Dropout doesn't rely on modifying the cost function.
- Instead, in dropout we modify the network itself. Let me describe the basic mechanics of how dropout works, before getting into why it works, and what the results are.
- Suppose we're trying to train a network:

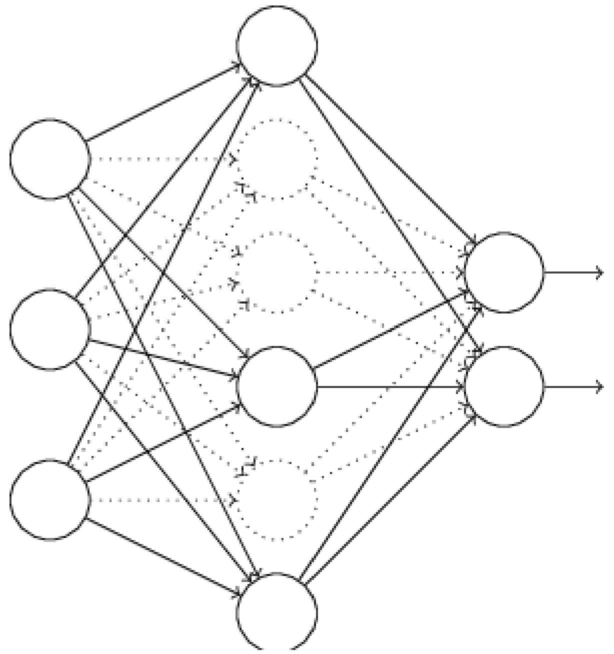


Dropout (cont..)

- In particular, suppose we have a training input x and corresponding desired output y.
- Ordinarily, we'd train by forward-propagating x through the network, and then backpropagating to determine the contribution to the gradient.
- With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched.
- After doing this, we'll end up with a network along the following lines. Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in:

Dropout (cont..)

- We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network.
- After doing this over a mini-batch of examples, we update the appropriate weights and biases.
- We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.



Dropout (cont..)

- By repeating this process over and over, our network will learn a set of weights and biases.
- Of course, those weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out.
- When we actually run the full network that means that twice as many hidden neurons will be active.
- To compensate for that, we halve the weights outgoing from the hidden neurons.

Why Dropout?

- This dropout procedure may seem strange and *ad hoc*.
- Imagine training neural networks in the standard way (no dropout). In particular, imagine we train several different neural networks, all using the same training data. Of course, the networks may not start out identical, and as a result after training they may sometimes give different results.
- When that happens we could use some kind of averaging or voting scheme to decide which output to accept. For instance, if we have trained five networks, and three of them are classifying a digit as a "3", then it probably really is a "3". The other two networks are probably just making a mistake.
- This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting. The reason is that the different networks may overfit in different ways, and averaging may help eliminate that kind of overfitting.

Why Dropout? (cont..)

- What's this got to do with dropout? Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks.
- And so the dropout procedure is like averaging the effects of a very large number of different networks.
- The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

Using Ensemble of Networks

- An easy way to improve performance still further is to create several neural networks, and then get them to vote to determine the best classification.
- Suppose, for example, that we trained 5 different neural networks using the prescription above, with each achieving accuracies near to 99.6 percent.
- Even though the networks would all have similar accuracies, they might well make different errors, due to the different random initializations.
- It's plausible that taking a vote amongst our 5 networks might yield a classification better than any individual network.

Softmax in the output layer

- The idea of softmax is to define a new type of output layer for our neural networks. It begins in the same way as with a sigmoid layer, by forming the weighted inputs:

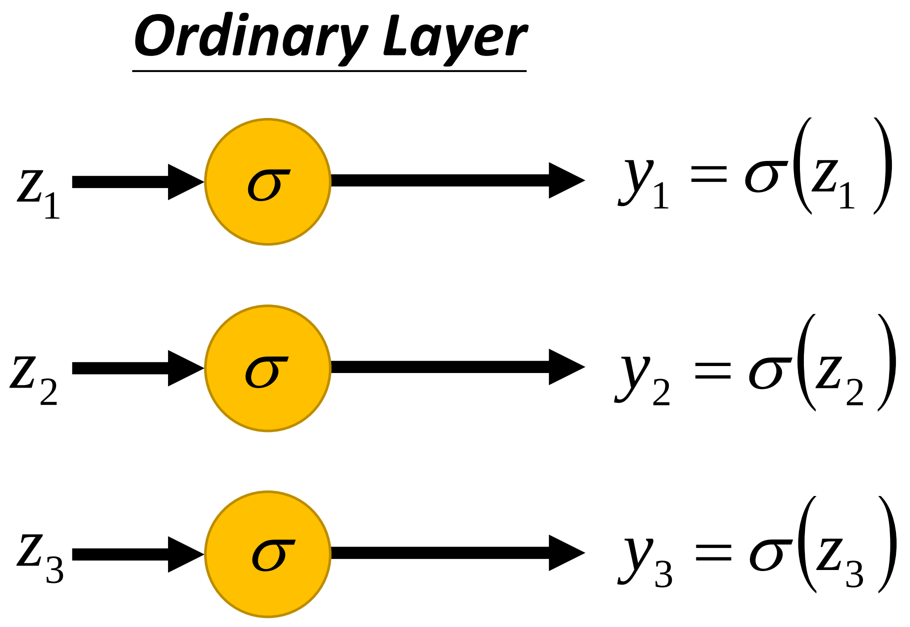
$$z^L_j = \sum_k w^L_{jk} a^{L-1}_k + b^L_j.$$

- However, we don't apply the sigmoid function to get the output. Instead, in a softmax layer we apply the so-called *softmax function* to the z^L_j.

- According to this function, the activation a_j^L of the jth output neuron is $$a^L_j = \frac{e^{z^L_j}}{\sum_k e^{z^L_k}},$$

Softmax in the output layer (Cont..)

- Softmax layer as the output layer



In general, the output of network can be any value.

May not be easy to interpret

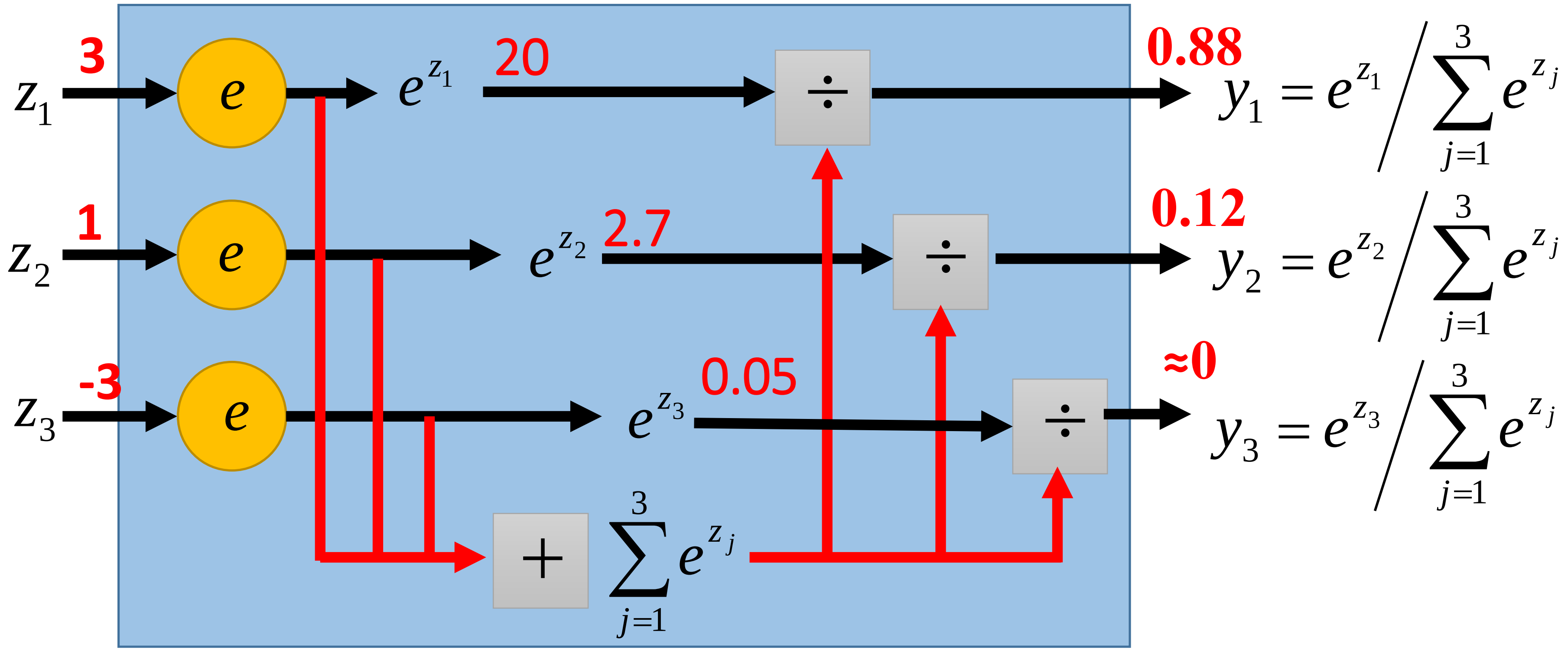
Softmax in the output layer (Cont..)

- Softmax layer as the output layer

Probability:

- $1 > y_i > 0$
- $\sum_i y_i = 1$

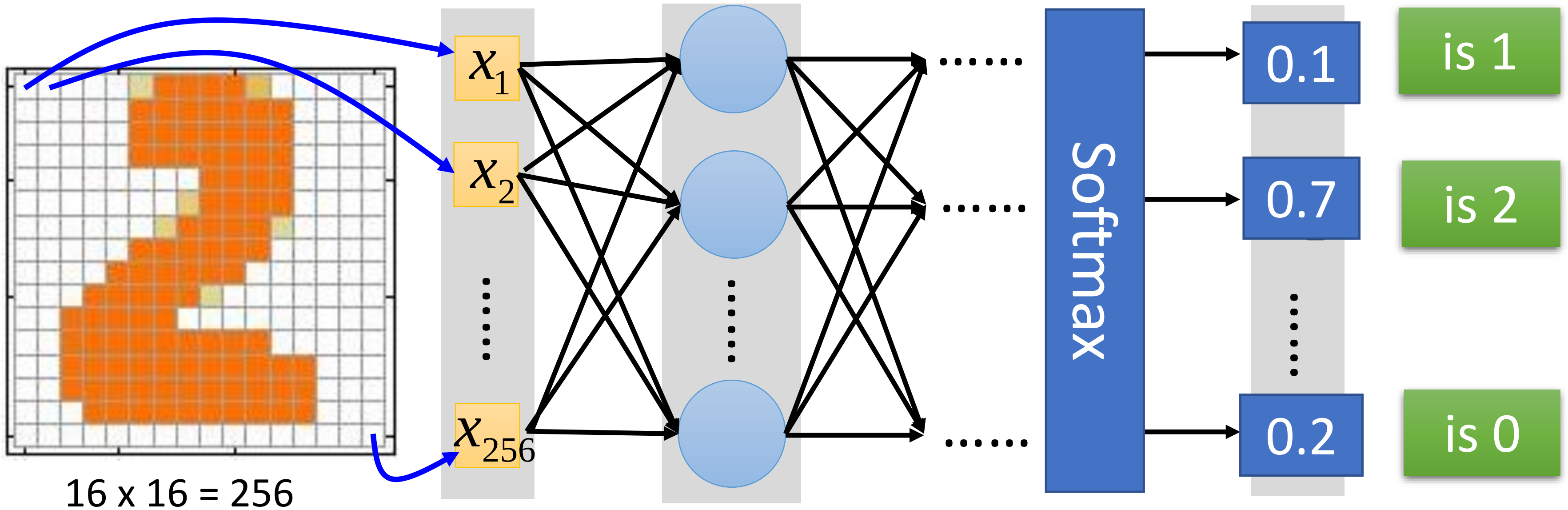
Softmax Layer



z_1
3
e
e^{z_1}
20
$\div$
0.88
$y_1 = e^{z_1} \Big/ \sum_{j=1}^{3} e^{z_j}$
z_2
1
e
e^{z_2}
2.7
$\div$
0.12
$y_2 = e^{z_2} \Big/ \sum_{j=1}^{3} e^{z_j}$
z_3
-3
e
e^{z_3}
0.05
$\div$
≈ 0
$y_3 = e^{z_3} \Big/ \sum_{j=1}^{3} e^{z_j}$
$+$
$\sum_{j=1}^{3} e^{z_j}$

Softmax in the output layer (Cont..)

$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$



16 x 16 = 256

Ink → 1

No ink → 0

Set the network parameters θ such that ……

Input m value

How to let the neural network achieve this

Input: y_2 has the maximum value

Problems in Quadratic Cost Function

- Recall that we're using the quadratic cost function, which, from Equation is given by:

$$C = \frac{(y-a)^2}{2},$$

- To write this more explicitly in terms of the weight and bias, a=σ(z)recall that, where z=wx+b. Using the chain rule to differentiate with respect to the weight and bias we get

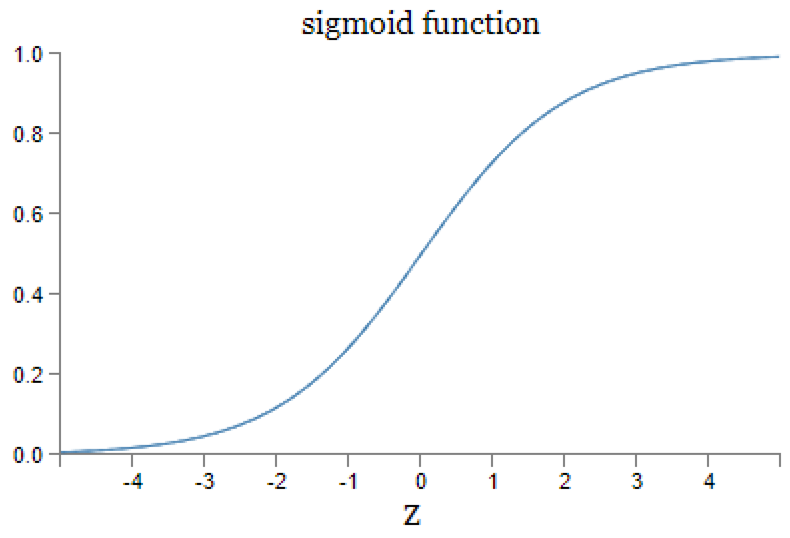
$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z),$$

- where I have substituted x=1 and y=0. To understand the behaviour of these expressions, let's look more closely at the σ′(z)term on the right-hand side. Recall the shape of the σ function:

Problems in Quadratic Cost Function (Cont..)

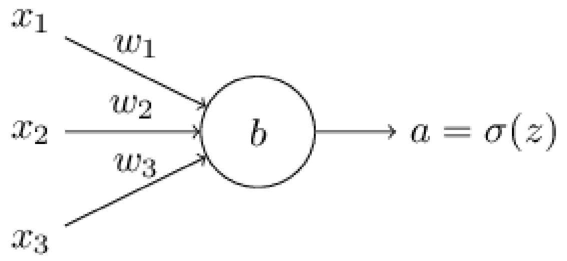
- We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, so $\sigma'(z)$ gets very small.

- Equations then tell us that $\partial C/\partial w$ and $\partial C/\partial b$ get very small. This is the origin of the learning slowdown.



Cross Entropy Cost Function

- How can we address the learning slowdown?
- It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the cross-entropy. To understand the cross-entropy,
- We'll suppose instead that we're trying to train a neuron with several input variables, x1,x2,...corresponding weights w1,w2,...and a bias, b. The output from the neuron is of course, a=σ(z), where $z = \sum_j w_j x_j + b$ is the weighted sum of the inputs. Cross Entropy C is defined as.



$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a)\right],$$

- where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

Cross Entropy Cost Function (Cont..)

- We know a=σ(z), let's compute the partial derivative of the cross-entropy cost with respect to the weights:

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)],$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$
$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j.$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y).$$

- Using the definition of the sigmoid function, $\sigma(z)=1/(1+e^{-z})$, and we have also shown that, first derivative σ' (z)=σ(z)(1−σ(z)), we see that the σ′(z) and σ(z)(1−σ(z)) terms cancel in the equation just above, and it simplifies to become $$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y).$$

Cross Entropy Cost Function (Cont..)

- This is a beautiful expression.
- It tells us that the rate at which the weight learns is controlled by $\sigma(z)-y$, i.e., by the error in the output.
- The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect.
- In particular, it avoids the learning slowdown caused by the $\sigma'(z)$term in the analogous equation for the quadratic cost, . When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out

Stochastic Gradient Descent and Mini Batch

- Cost function has the form that is, it's an average $C = \frac{1}{n}\sum_x C_x$ over costs for individual training examples, where

$$C_x \equiv \frac{\|y(x)-a\|^2}{2}$$

- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x.$$

- Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly
- An idea called *stochastic gradient descent* can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇Cx for a small sample of randomly chosen training inputs.
- By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

Stochastic Gradient Descent and Mini Batch (cont..)

- To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs X1,X2,...,Xm, and refer to them as a *mini-batch*.

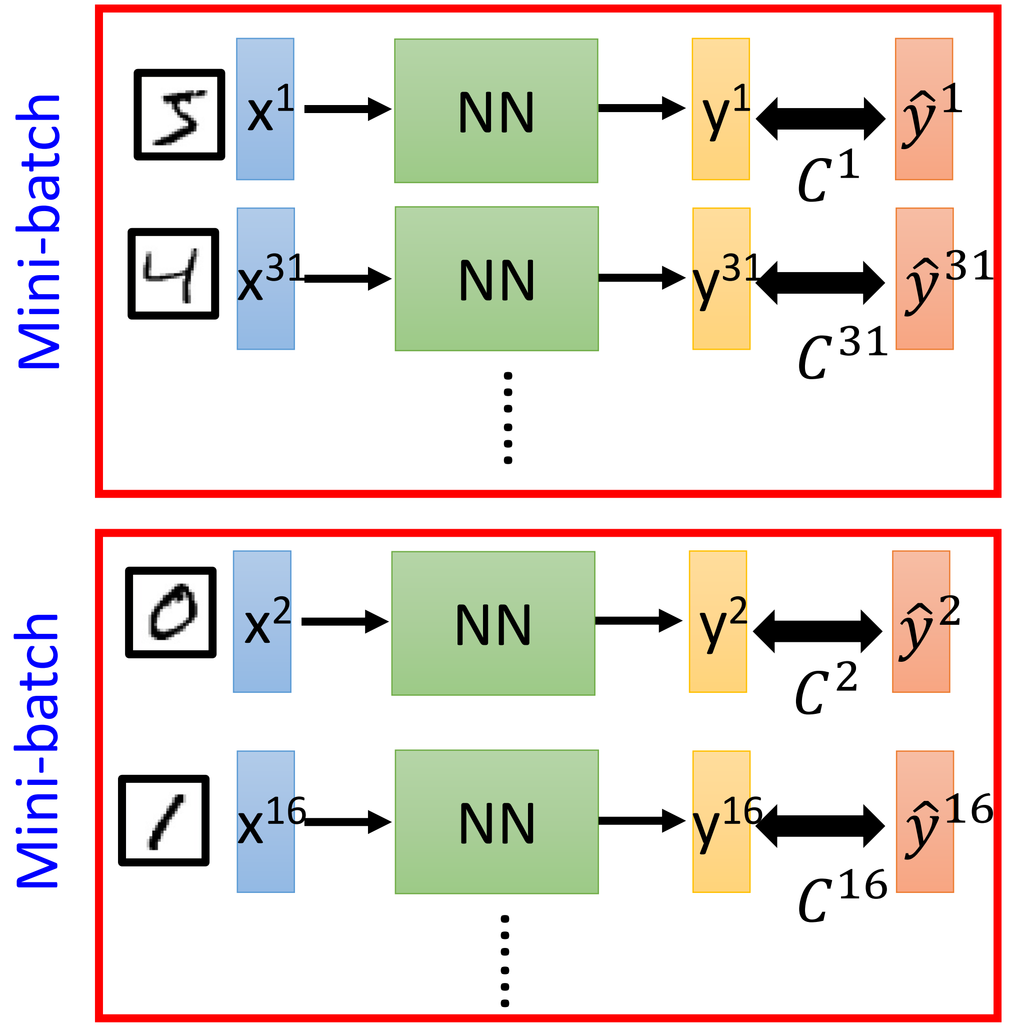
- Provided the sample size m is large enough we expect that the average value of the ∇C_{X_j} will be roughly equal to the average over all ∇C_x, that is:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

Mini-batch

Faster

Better!



- Randomly initialize θ^0
- Pick the 1st batch

  $C = C^1 + C^{31} + \cdots$

  $\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$
- Pick the 2nd batch

  $C = C^2 + C^{16} + \cdots$

  $\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$

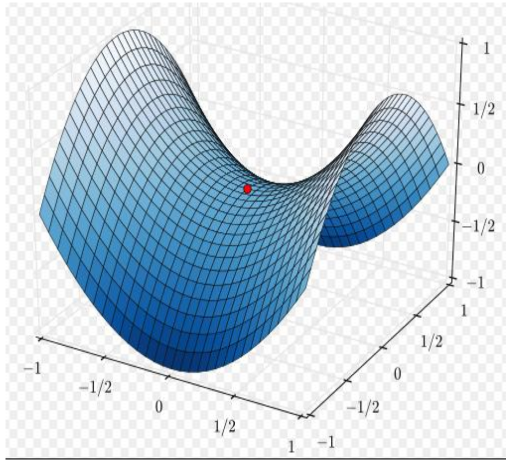
  ⋮
- Until all mini-batches have been picked

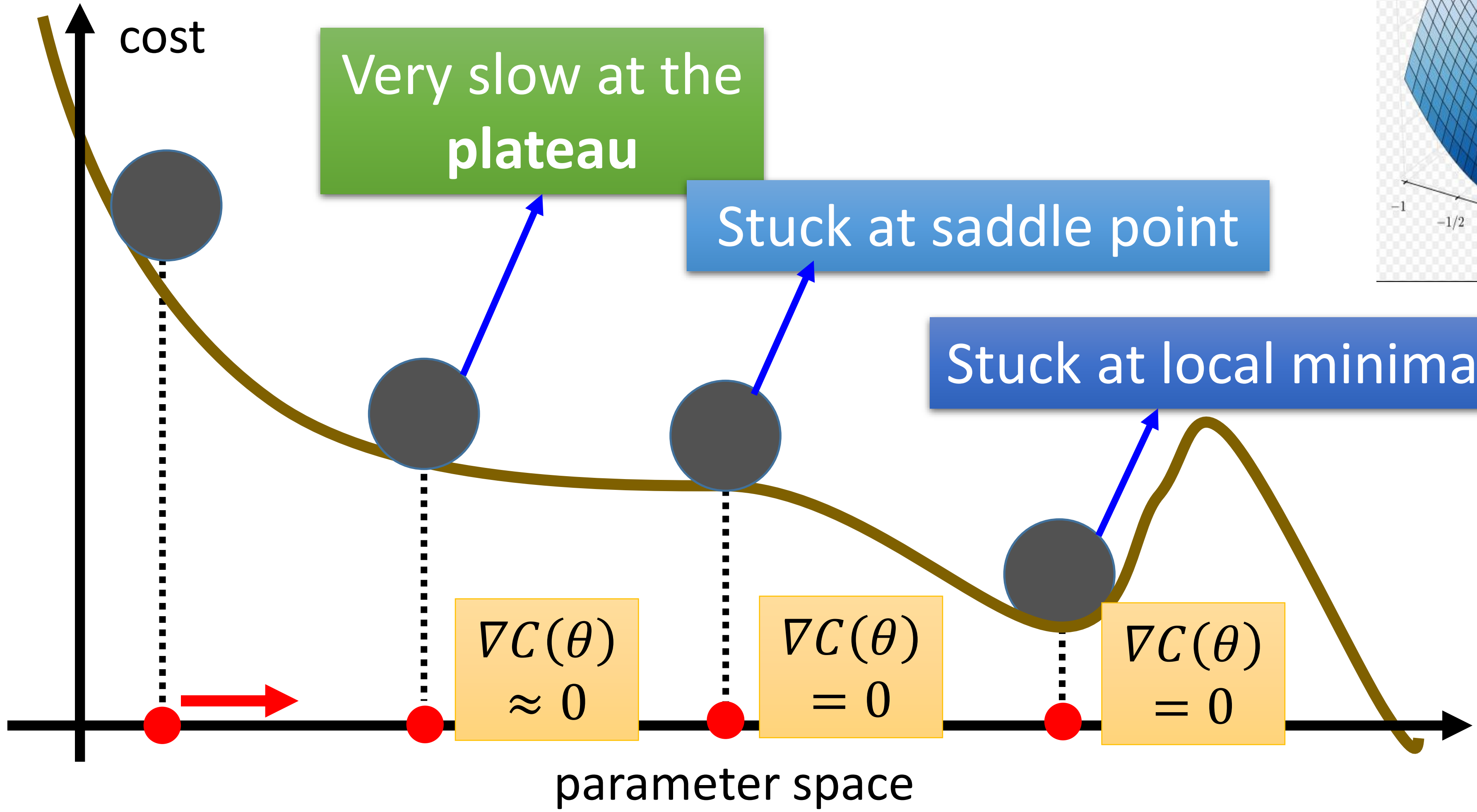
one epoch

Repeat the above process

Besides local minima ……

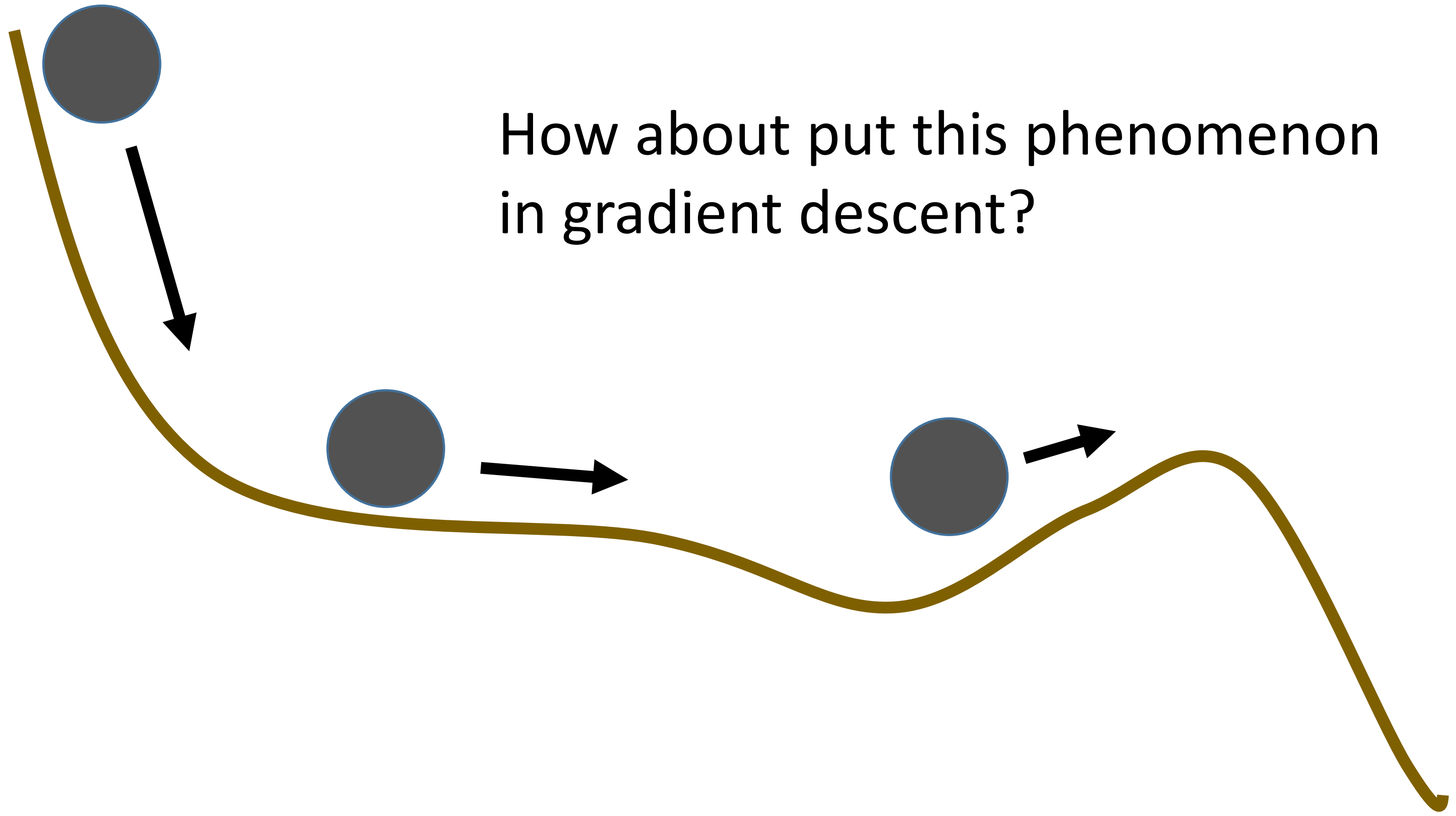
In mathematics, a **saddle point** or minimax **point** is a **point** on the surface of the graph of a function where the slopes (derivatives) in orthogonal directions are both zero (a critical **point**), but which is not a **local** extremum of the function.





In physical world ……

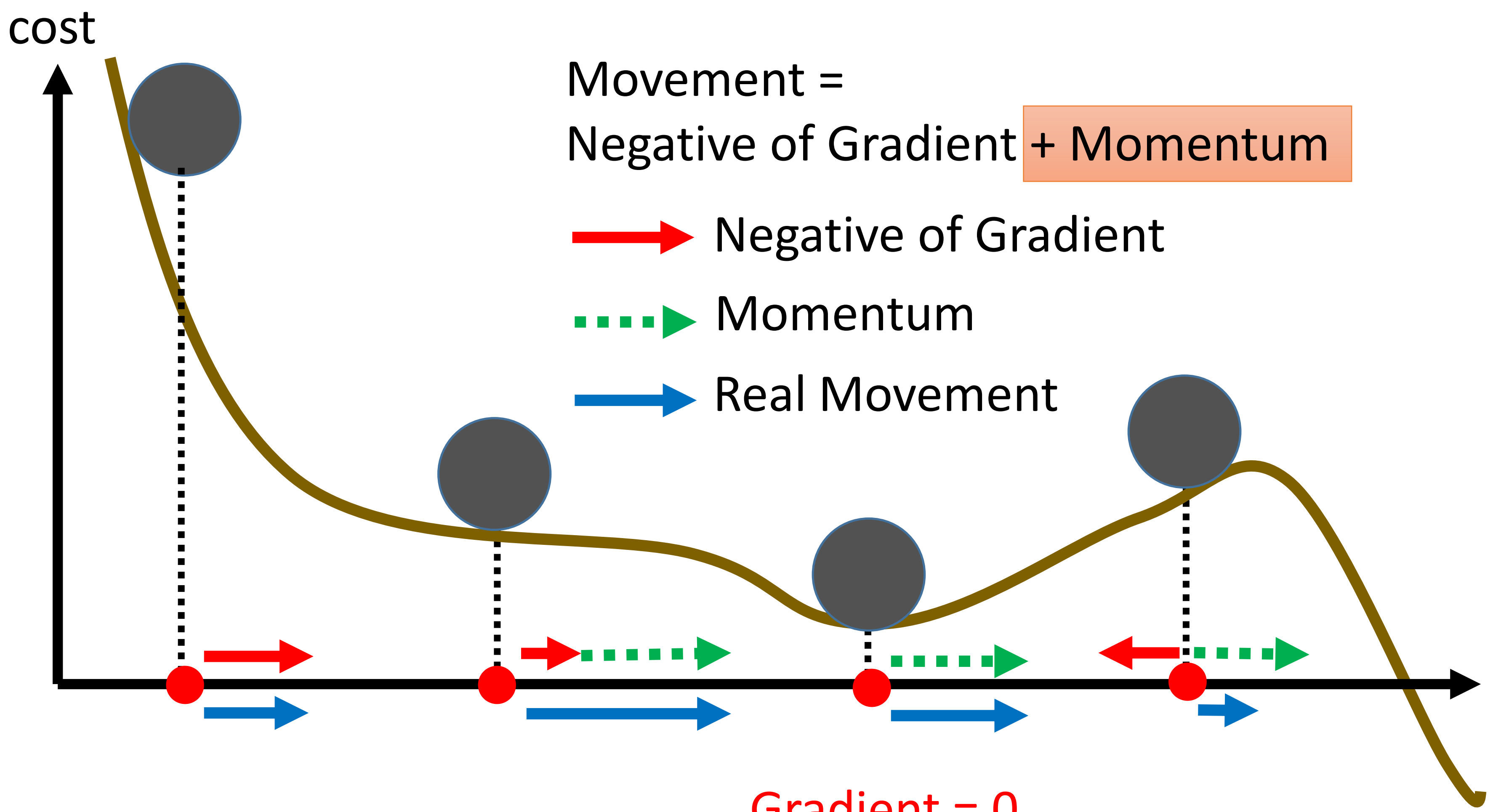
- Momentum



How about put this phenomenon in gradient descent?

Momentum

Still not guarantee reaching global minima, but give some hope ……



Movement =
Negative of Gradient + Momentum

→ Negative of Gradient

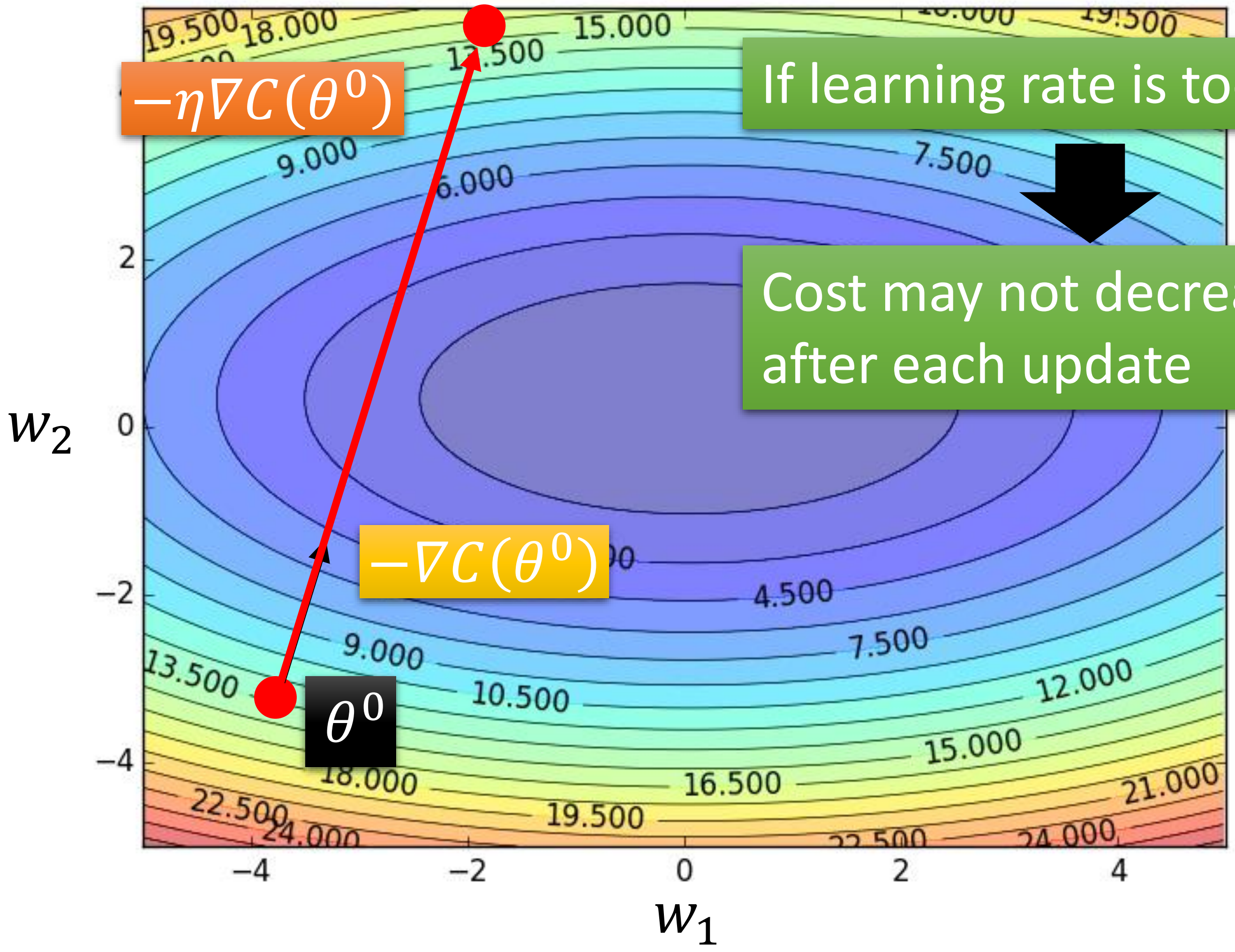
⇢ Momentum

→ Real Movement

Gradient = 0

Learning Rate

Set the learning rate η carefully

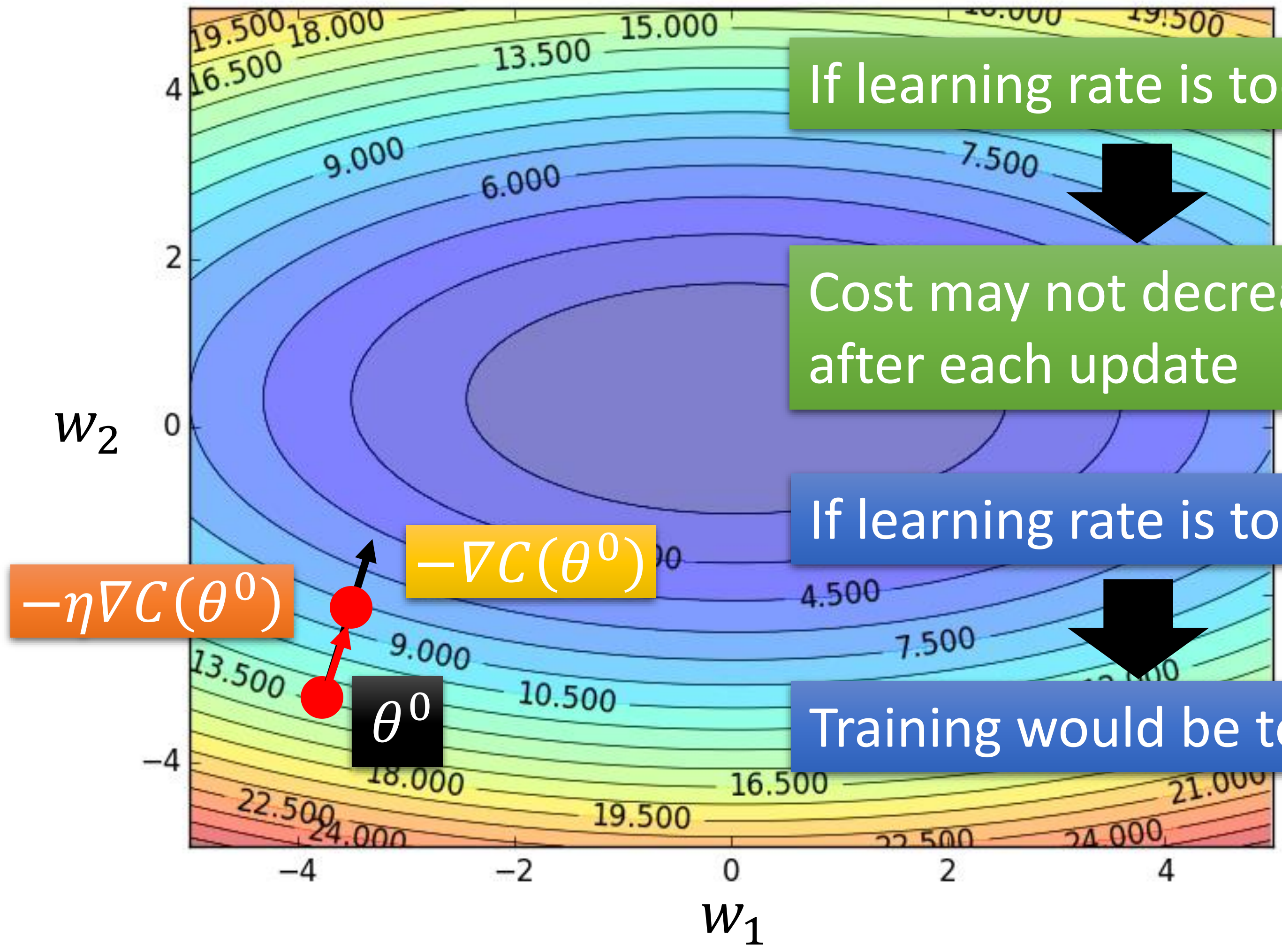


If learning rate is too large

Cost may not decrease after each update

Learning Rate

Can we give different parameters different learning rates?



If learning rate is too large

Cost may not decrease after each update

If learning rate is too small

Training would be too slow

Adagrad

Original Gradient Descent

$$\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

Each parameter w are considered separately

$$w^{t+1} \leftarrow w^t - \boxed{\eta_w} \underline{g^t} \qquad \underline{g^t} = \frac{\partial C(\theta^t)}{\partial w}$$

Parameter dependent learning rate

$$\eta_w = \frac{\eta}{\sqrt{\boxed{\sum_{i=0}^{t} (g^i)^2}}}$$

constant

Summation of the square of the previous derivatives

Adagrad

$$\eta_w = \boxed{\frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}}}$$

	g^0
w_1	0.1

	g^0
w_2	20.0

Learning rate: | Learning rate:

$$\begin{array}{ccccc} \frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1} & & \longleftrightarrow & \frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20} & \\ & \Big\downarrow & & & \Big\downarrow \\ \frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22} & & \longleftrightarrow & \frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22} & \end{array}$$

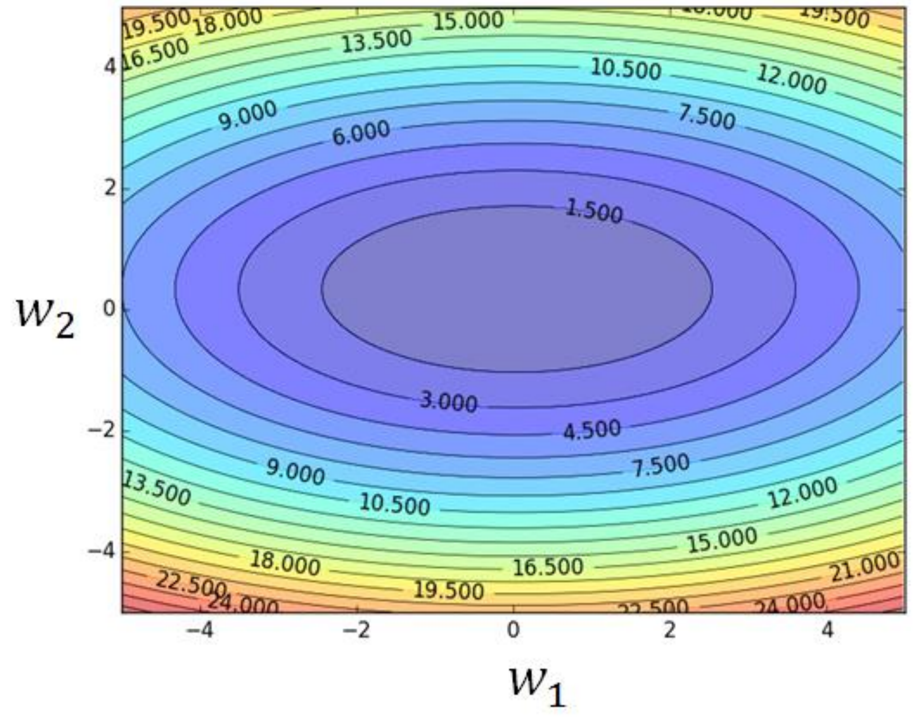
Observation: 1. Learning rate is smaller and smaller for all parameters

2. Smaller derivatives, larger learning rate, and vice versa

Why?

Larger derivatives

Smaller Learning Rate



Smaller Derivatives

Larger Learning Rate

2. Smaller derivatives, larger learning rate, and vice versa

Why?

Not the whole story ......

- Adagrad [John Duchi, JMLR'11]
- RMSprop
  - https://www.youtube.com/watch?v=O3sxAc4hxZU
- Adadelta [Matthew D. Zeiler, arXiv'12]
- Adam [Diederik P. Kingma, ICLR'15]
- AdaSecant [Caglar Gulcehre, arXiv'14]
- "No more pesky learning rates" [Tom Schaul, arXiv'12]