Neural Networks

Dr Amit Sethi, IITB

Module objectives

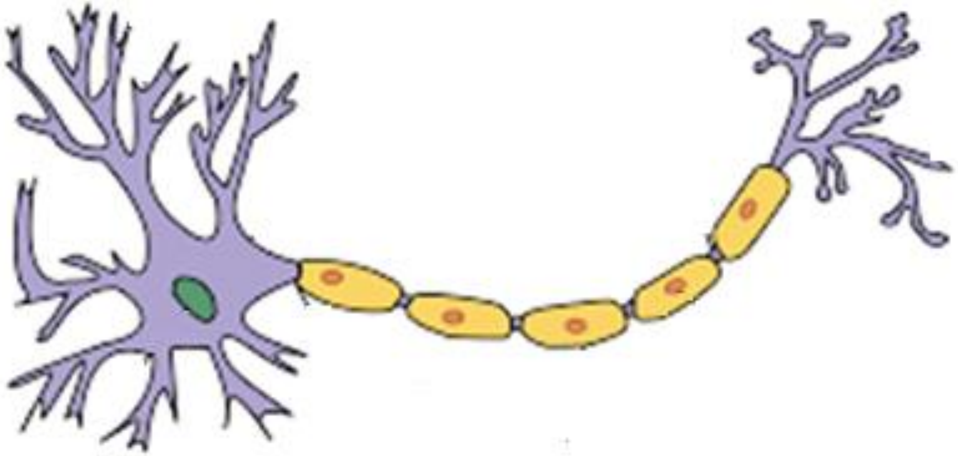
- Learn the motivations behind neural networks
- Become familiar with neural network terms
- Understand the working of neural networks
- Understand behind-the-scenes training of neural networks

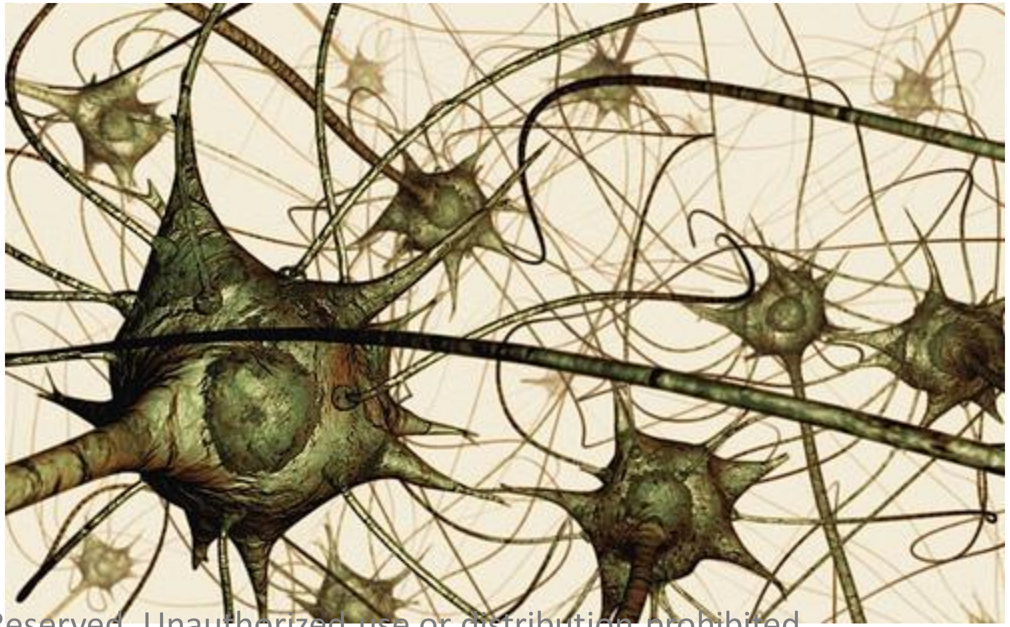
Contents

- Introduction to neural networks
- Feed forward neural networks
- Gradient descent and backpropagation
- Learning rate setting and tuning

Neural networks are inspired from mammalian brain

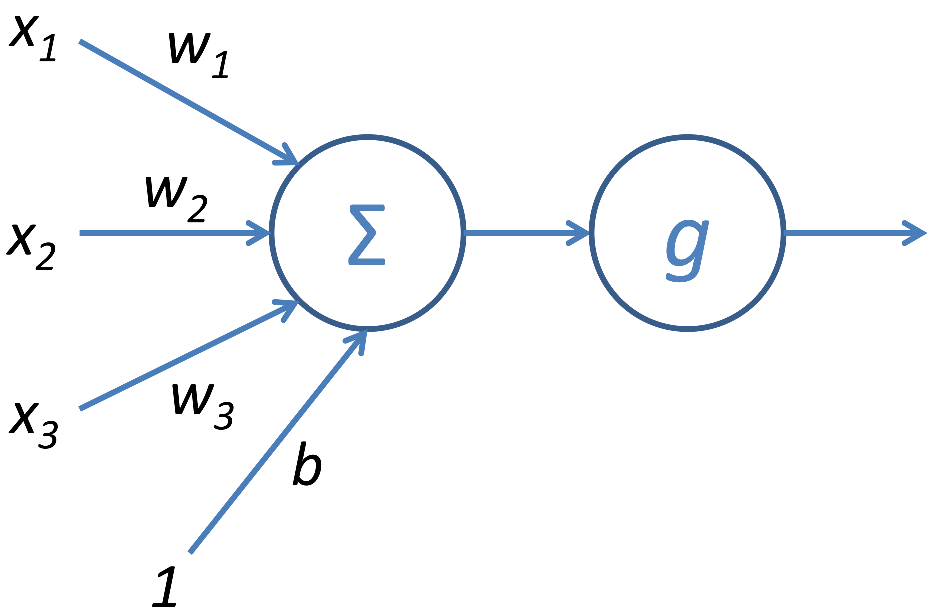
- Each unit (neuron) is simple
- But, human brain has 100 billion neurons with 100 trillion connections
- The strength and nature of the connections stores memories and the "program" that makes us human
- A neural network is a web of artificial neurons



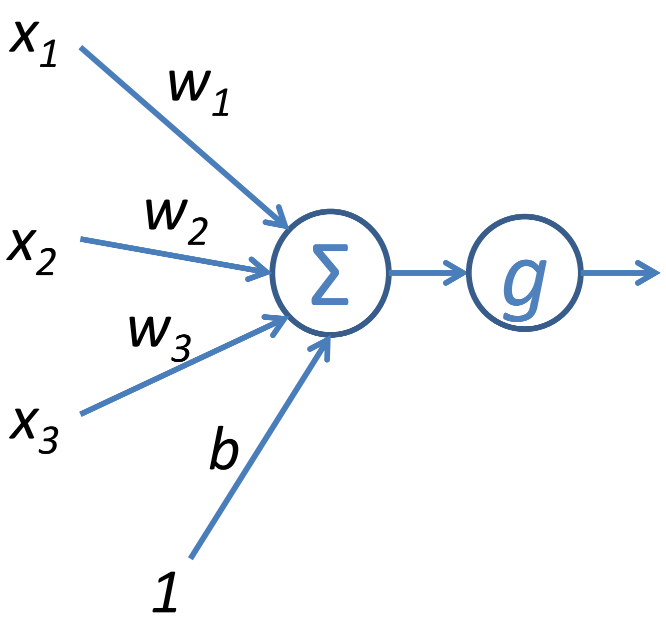


Artificial neurons is inspired by biological neurons

- Neural networks are made up of artificial neurons
- Artificial neurons are only loosely based on real neurons, just like neural networks are only loosely based on the human brain

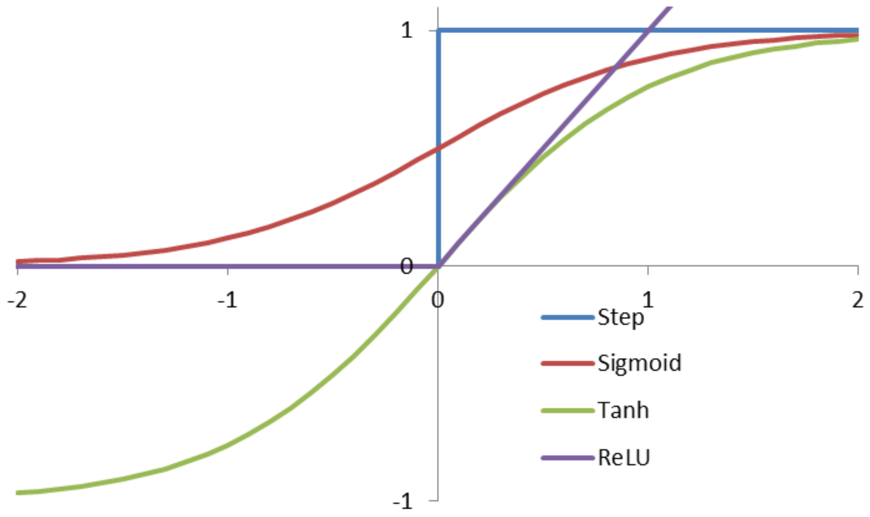


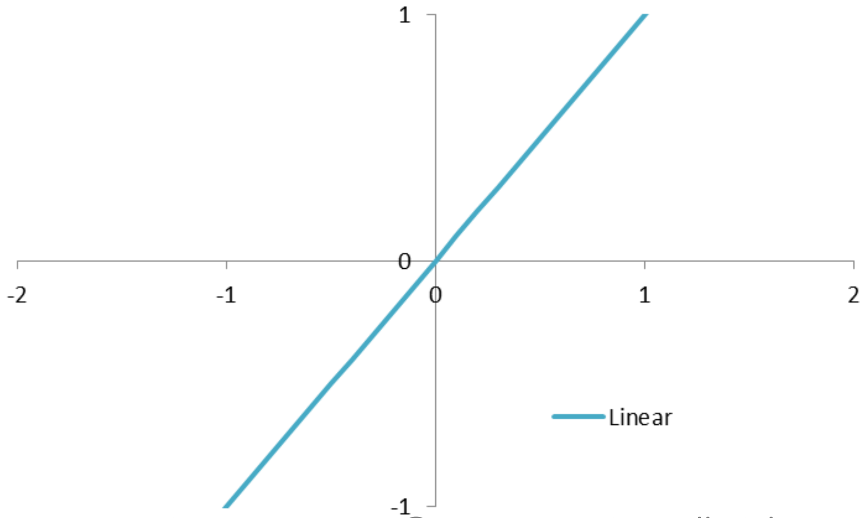
Activation function is the secret sauce of neural networks



- Neural network training is all about tuning weights and biases
- If there was no activation function *f*, the output of the entire neural network would be a linear function of the inputs
- In a perceptron, it was a step function

Types of activation functions

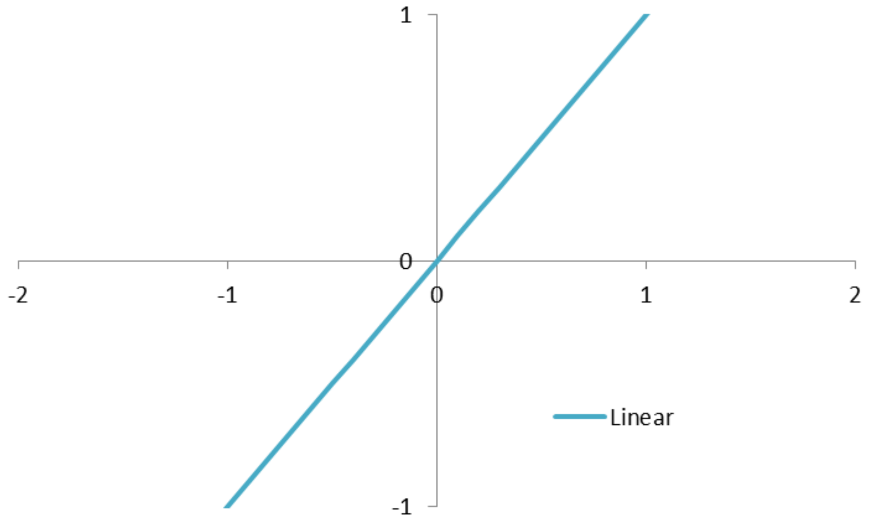




- **Step**: original concept behind classification and region bifurcation. Not used anymore
- **Sigmoid** and tanh: trainable approximations of the step-function
- **ReLU**: currently preferred due to fast convergence
- **Softmax**: currently preferred for output of a *classification net.* Generalized sigmoid
- **Linear**: good for modeling a range in the output of a *regression net*

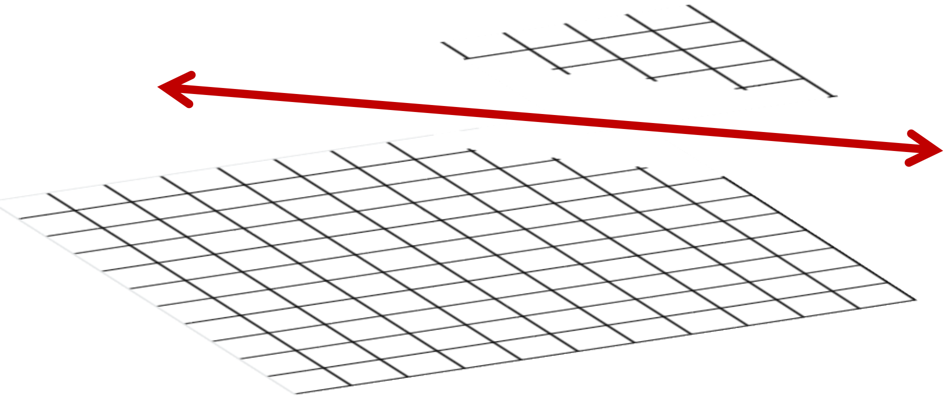
Formulas for activation functions

- **Step**: $g(x) = \frac{\text{sign}(x)+1}{2}$
- **Sigmoid**: $g(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $g(x) = \tanh(x)$
- **ReLU**: $g(x) = \max(0, x)$
- **Softmax**: $g(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$
- **Linear**: $g(x) = x$



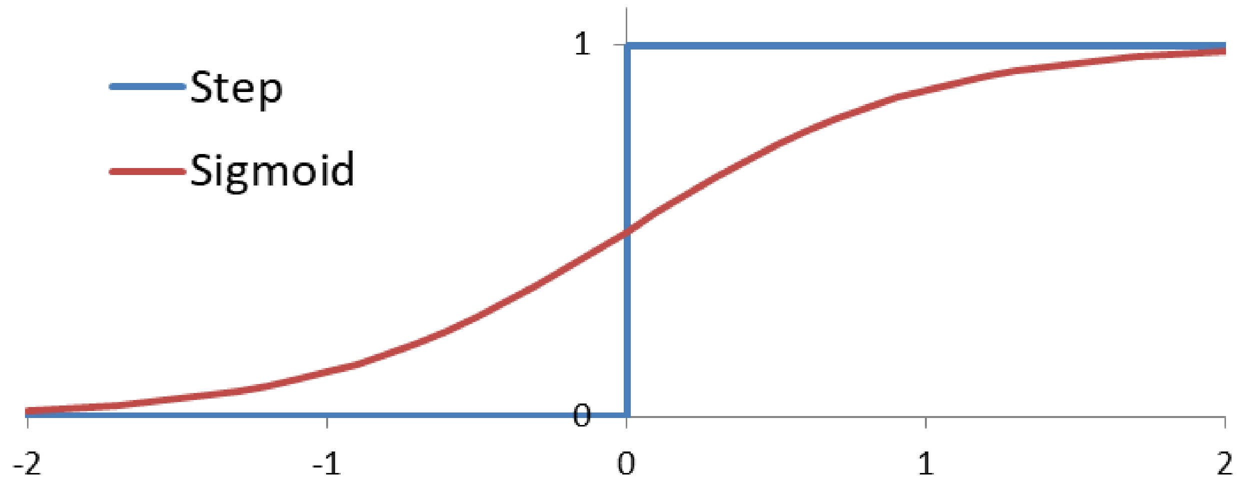
- **Step**: $g(x) = \frac{\text{sign}(x)+1}{2}$
- **Sigmoid**: $g(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $g(x) = \tanh(x)$
- **ReLU**: $g(x) = \max(0, x)$
- **Softmax**: $g(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$
- **Linear**: $g(x) = x$

Step function divides the input space into two halves → 0 and 1



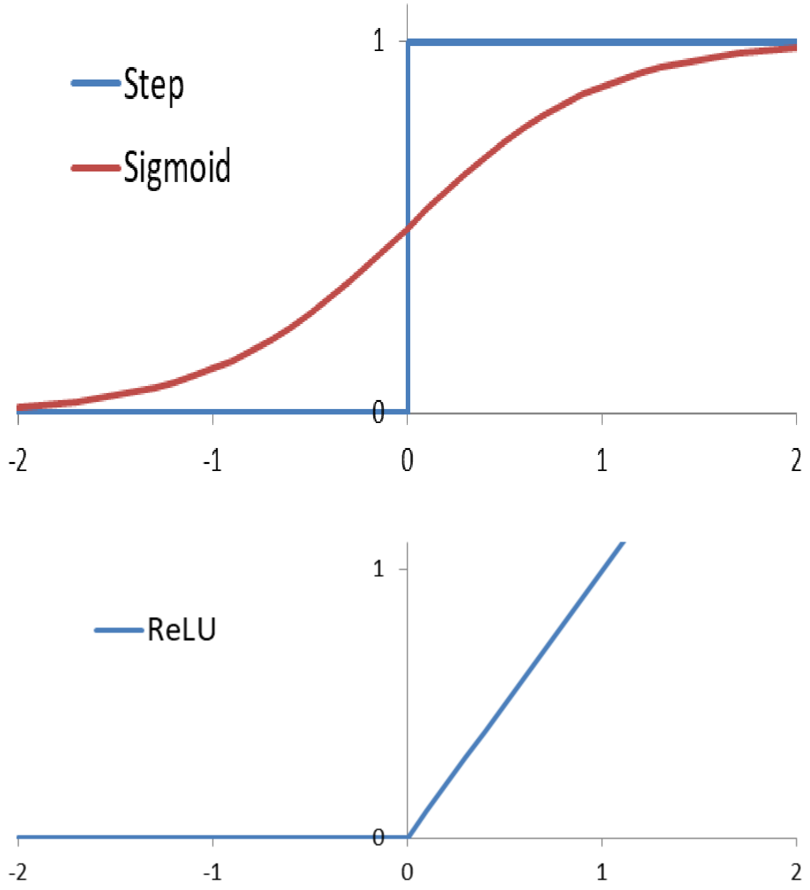
- In a single neuron, step function is a linear binary classifier
- The weights and biases determine where the step will be in n-dimensions
- But, as we shall see later, it gives little information about how to change the weights if we make a mistake
- So, we need a smoother version of a step function
- Enter: the Sigmoid function

The sigmoid function is a smoother step function



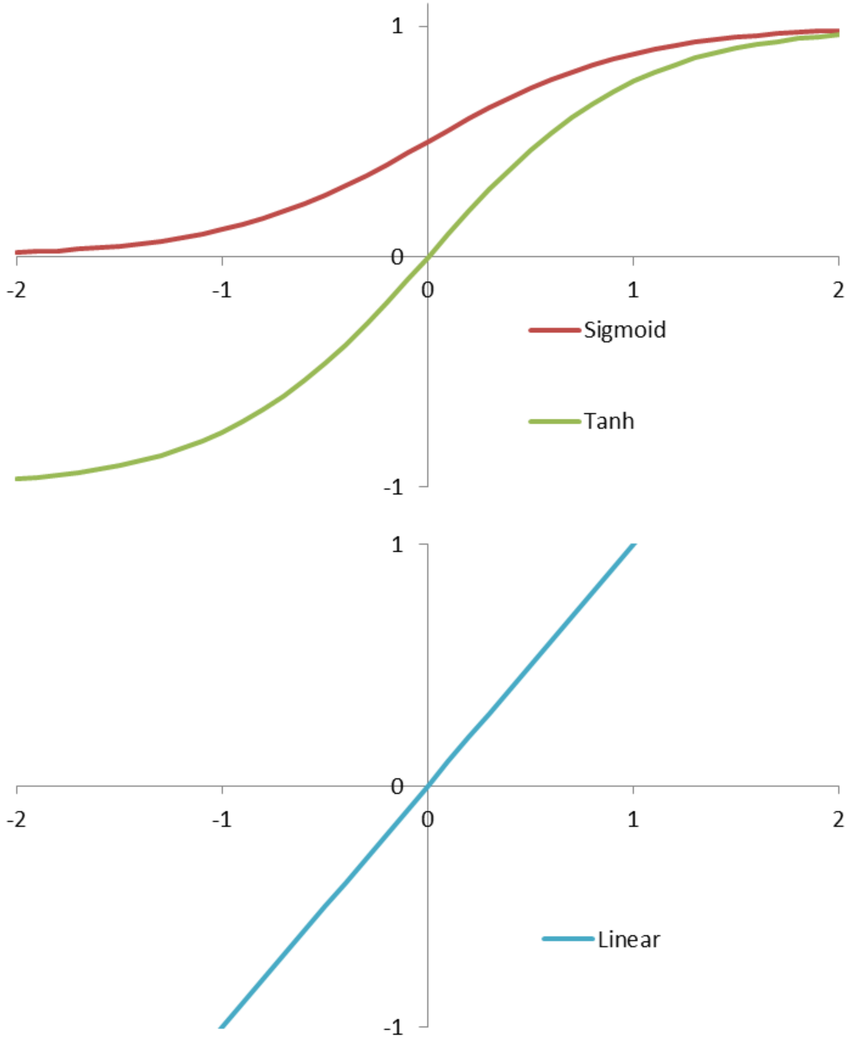
- Smoothness ensures that there is more information about the direction in which to change the weights if there are errors
- Sigmoid function is also mathematically linked to logistic regression, which is a theoretically well-backed linear classifier

The problem with sigmoid is (near) zero gradient on both extremes



- For both large positive and negative input values, sigmoid doesn’t change much with change of input
- ReLU has a constant gradient for almost half of the inputs
- But, ReLU cannot give a meaningful final output

Output activation functions can only be of the following kinds

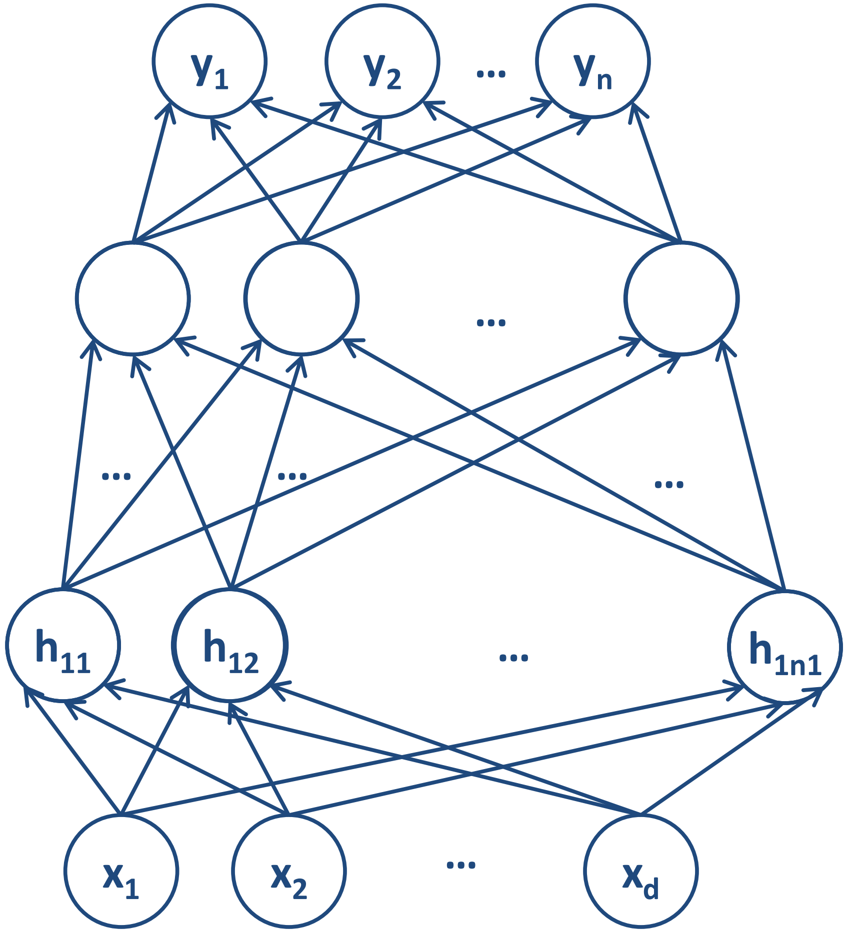


- Sigmoid gives binary classification output
- Tanh can also do that provided the desired output is in {-1, +1}
- Softmax generalizes sigmoid to n-ary classification
- Linear is used for regression
- ReLU is only used in internal nodes (non-output)

Contents

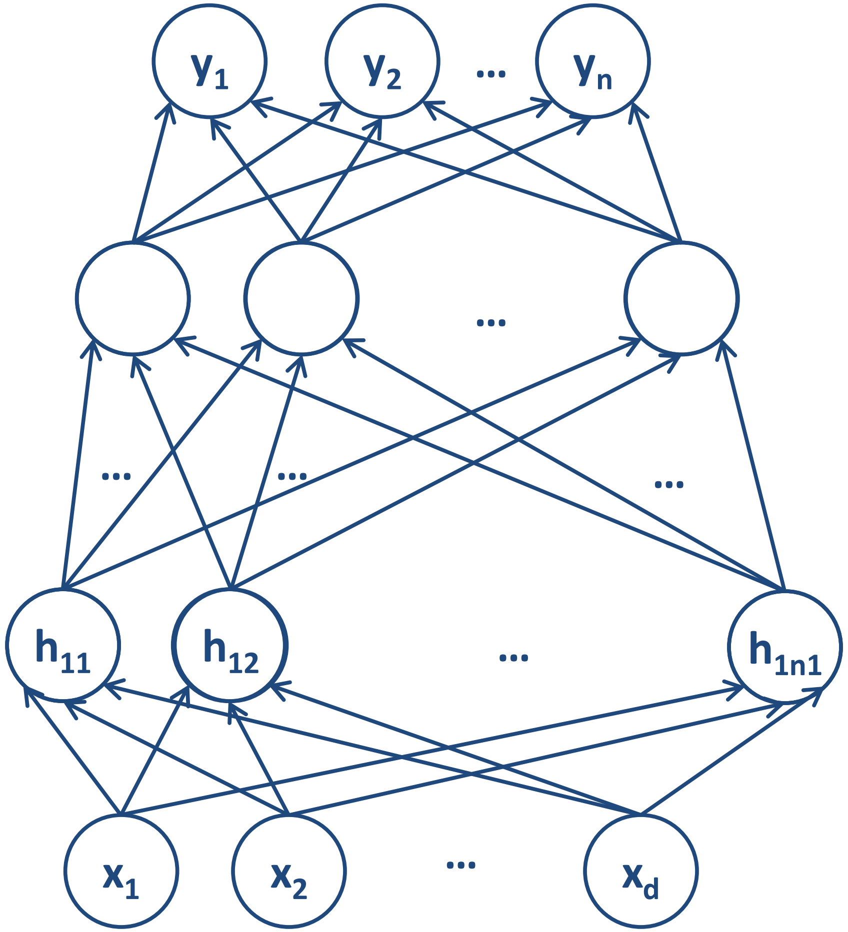
- Introduction to neural networks
- Feed forward neural networks
- Gradient descent and backpropagation
- Learning rate setting and tuning

Basic structure of a neural network



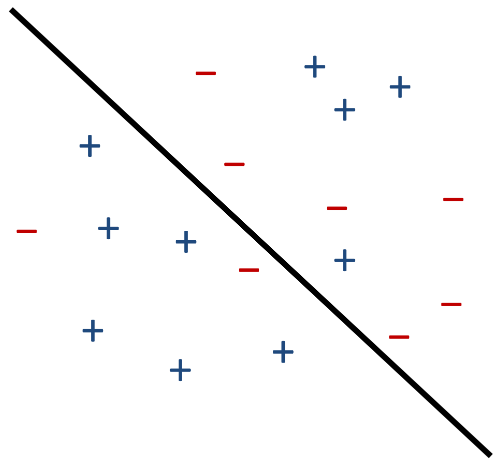
- It is feed forward
  - Connections from inputs towards outputs
  - No connection comes backwards
- It consists of layers
  - Current layer’s input is previous layer’s output
  - No lateral (intra-layer) connections
- That’s it!

Basic structure of a neural network

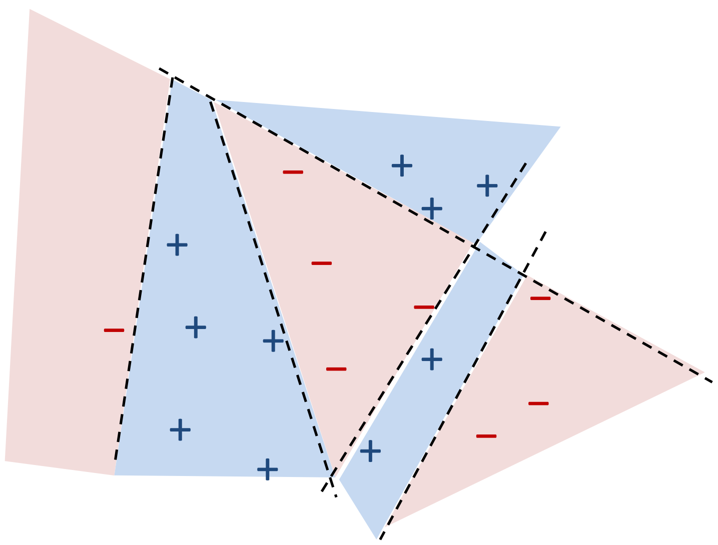


- **Output layer**
  - Represent the output of the neural network
  - For a two class problem or regression with a 1-d output, we need only one output node
- **Hidden layer(s)**
  - Represent the intermediary nodes that divide the input space into regions with (soft) boundaries
  - These usually form a *hidden layer*
  - Usually, there is only one such layer
  - Given enough hidden nodes, we can model an arbitrary input-output relation.
- **Input layer**
  - Represent dimensions of the input vector (one node for each dimension)
  - These usually form an *input layer*, and
  - Usually there is only one such layer

Importance of hidden layers



Single sigmoid



Sigmoid hidden layers and sigmoid output

- First hidden layer extracts features
- Second hidden layer extracts features of features
- …
- Output layer gives the desired output

Overall function of a neural network

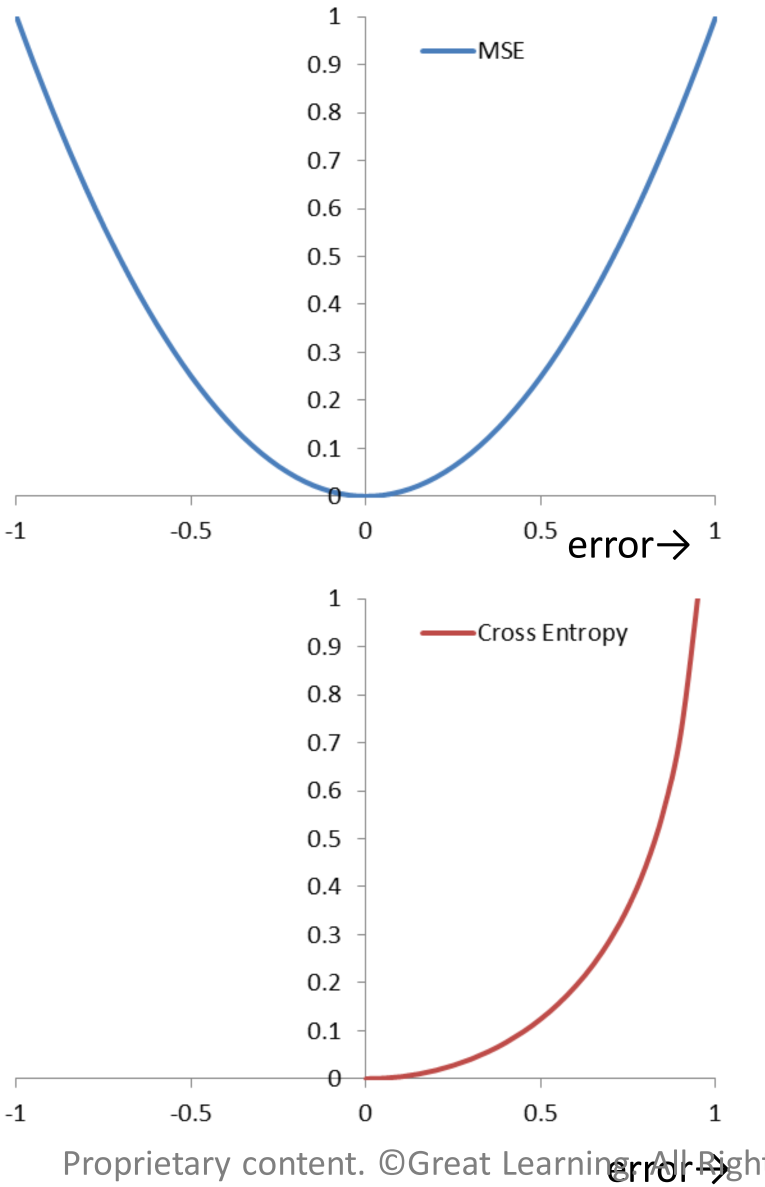
- $f(\boldsymbol{x}_i) = g_l(\boldsymbol{W}_l * g_{l-1}(\boldsymbol{W}_{l-1} \dots g_1(\boldsymbol{W}_1 * \boldsymbol{x}_i) \dots))$
- Weights form a matrix
- Output of the previous layer form a vector
- The activation (nonlinear) function is applied point-wise to the weight times input

- Design questions (hyper parameters):
  - Number of layers
  - Number of neurons in each layer (rows of weight matrices)

Training the neural network

- Given $\boldsymbol{x}_i$ and y_i
- Think of what hyper-parameters and neural network design might work
- Form a neural network: $f(\boldsymbol{x}_i) = g_l(\boldsymbol{W}_l * g_{l-1}(\boldsymbol{W}_{l-1} \ldots g_1(\boldsymbol{W}_1 * \boldsymbol{x}_i) \ldots))$
- Compute $f_{\boldsymbol{w}}(\boldsymbol{x}_i)$ as an estimate of y_i for all samples
- Compute loss: $\frac{1}{N}\sum_{i=1}^{N} L(f_{\boldsymbol{w}}(\boldsymbol{x}_i), y_i) = \frac{1}{N}\sum_{i=1}^{N} l_i(\boldsymbol{w})$
- Tweak $\boldsymbol{w}$ to reduce loss (optimization algorithm)
- Repeat last three steps

Loss function choice

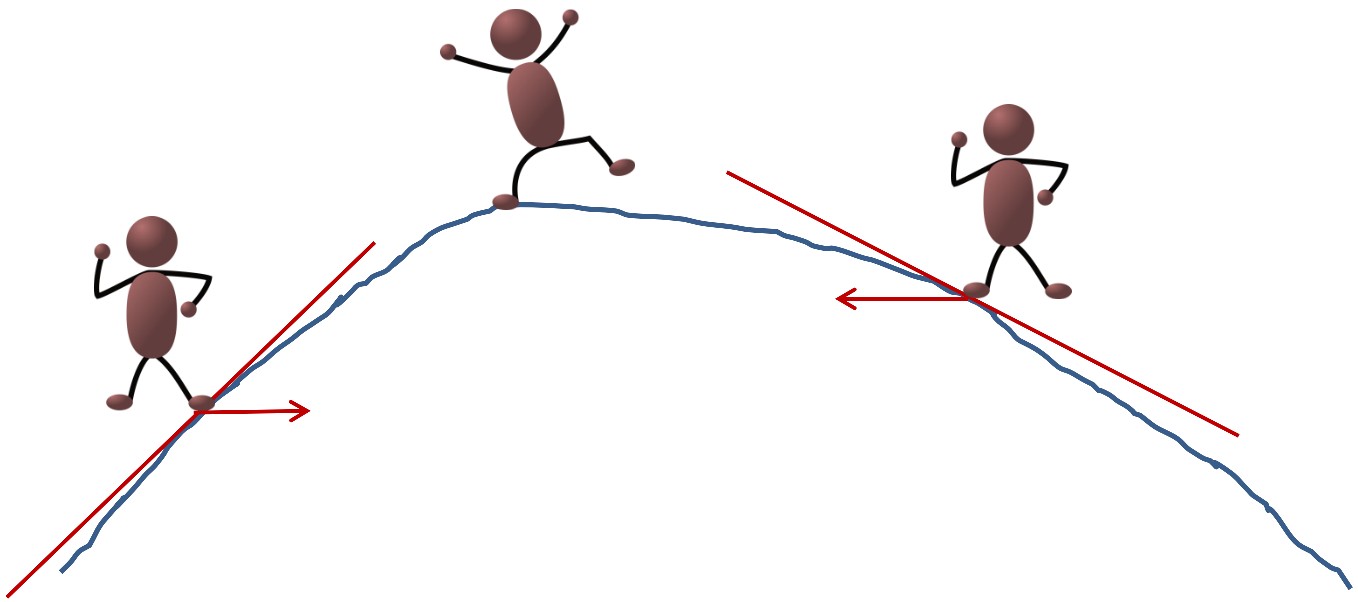


- There are positive and negative errors in classification and MSE is the most common loss function
- There is probability of correct class in classification, for which cross entropy is the most common loss function

Contents

Gradient ascent

- If you didn't know the shape of a mountain
- But at every step you knew the slope
- Can you reach the top of the mountain?



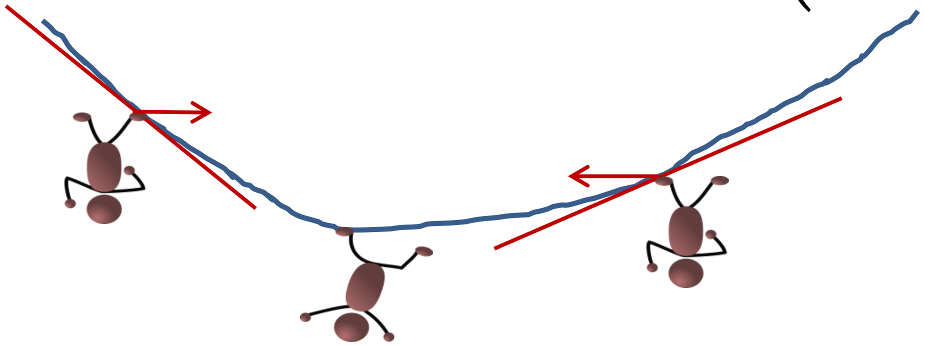
Gradient descent minimizes the loss function

- At every point, compute
  - Loss (scalar): $l_i(\boldsymbol{w})$
  - Gradient of loss with respect to weights (vector):

$$\nabla_{\boldsymbol{w}} l_i(\boldsymbol{w})$$

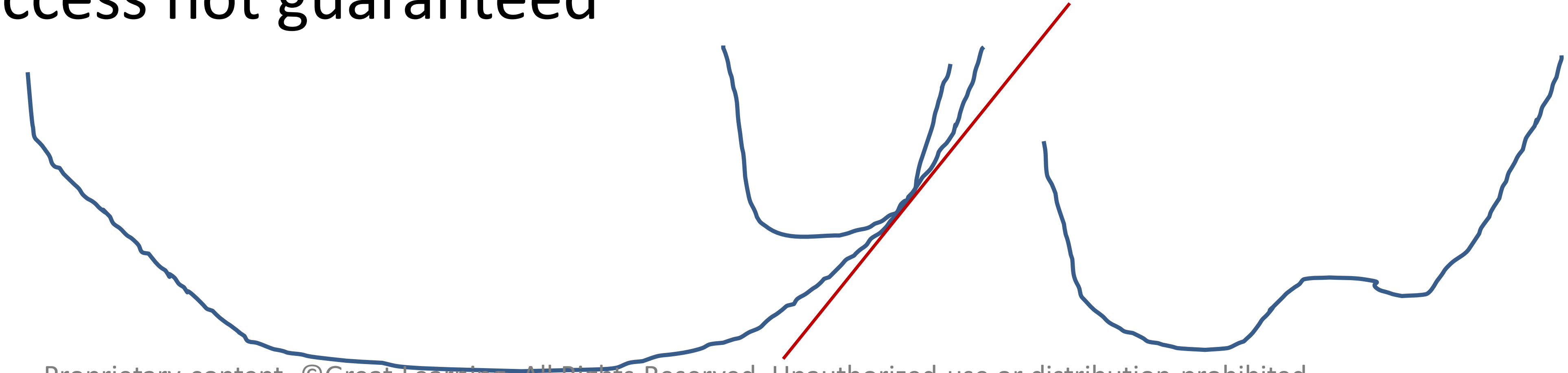
  - Take a step towards negative gradient:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \, \nabla_{\boldsymbol{w}} \left(\frac{1}{N} \sum_{i=1}^{N} l_i(\boldsymbol{w}) \right)$$



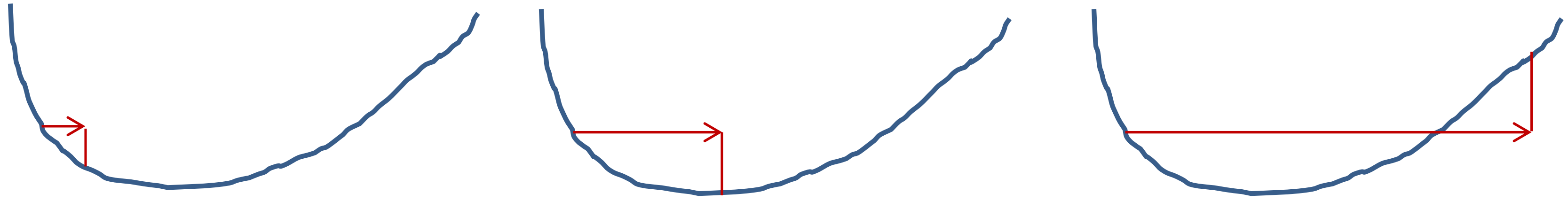
Role of step size and learning rate

- Tale of two loss functions
  - Same value, and
  - Same gradient (first derivative), but
  - Different Hessian (second derivative)
  - Different step sizes needed
- Success not guaranteed



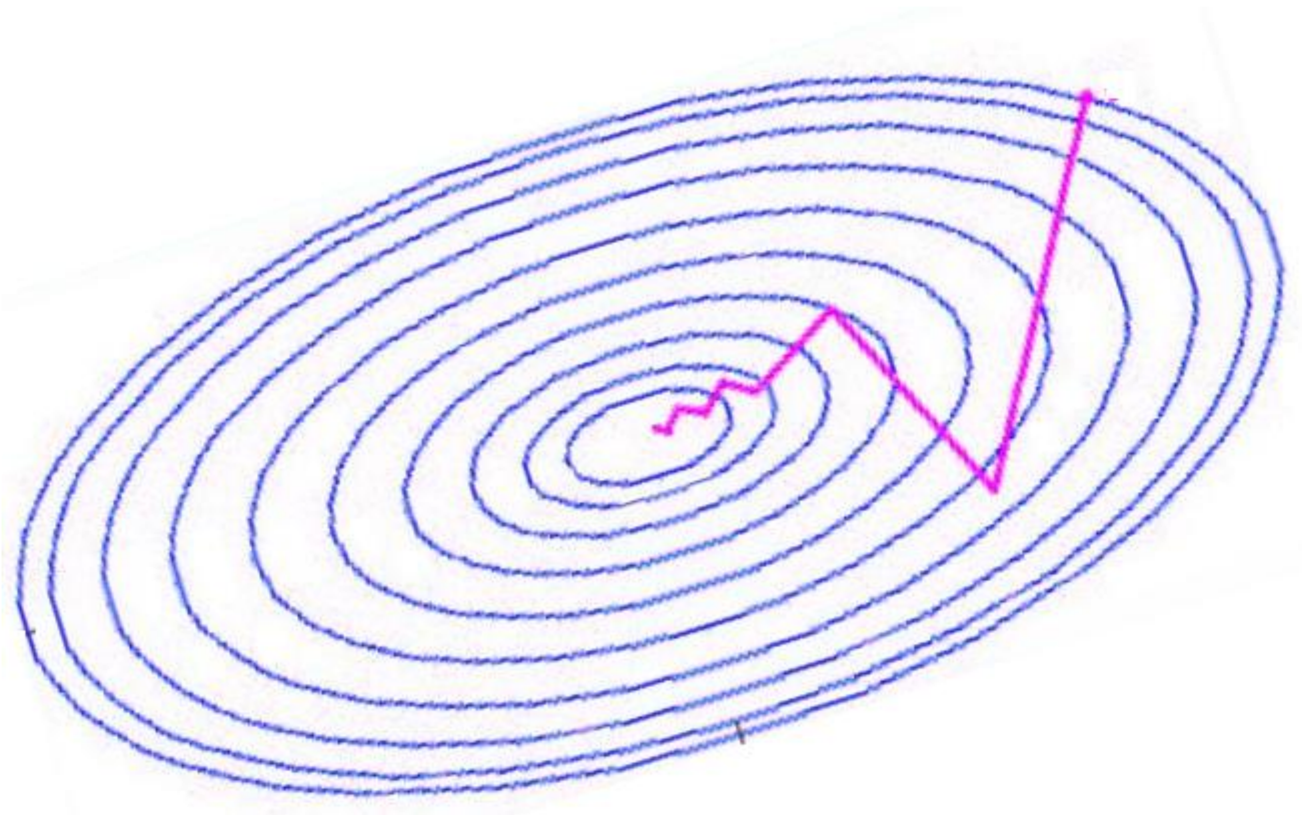
The perfect step size is impossible to guess

- Goldilocks finds the perfect balance only in a fairy tale

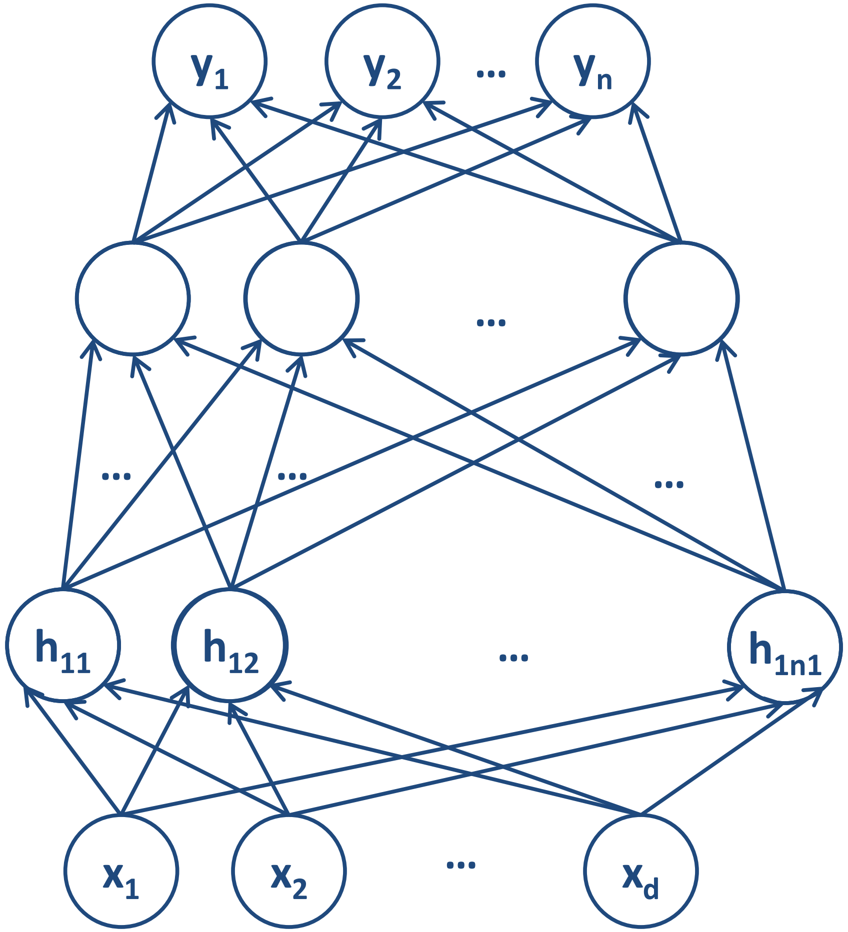


- The step size is decided by learning rate η and the gradient

This story is unfolding in multiple dimensions



Backpropagation



- Backpropagation is an efficient method to do gradient descent
- It saves the gradient w.r.t. the upper layer output to compute the gradient w.r.t. the weights immediately below
- It is linked to the chain rule of derivatives
- All intermediary functions must be differentiable, including the activation functions

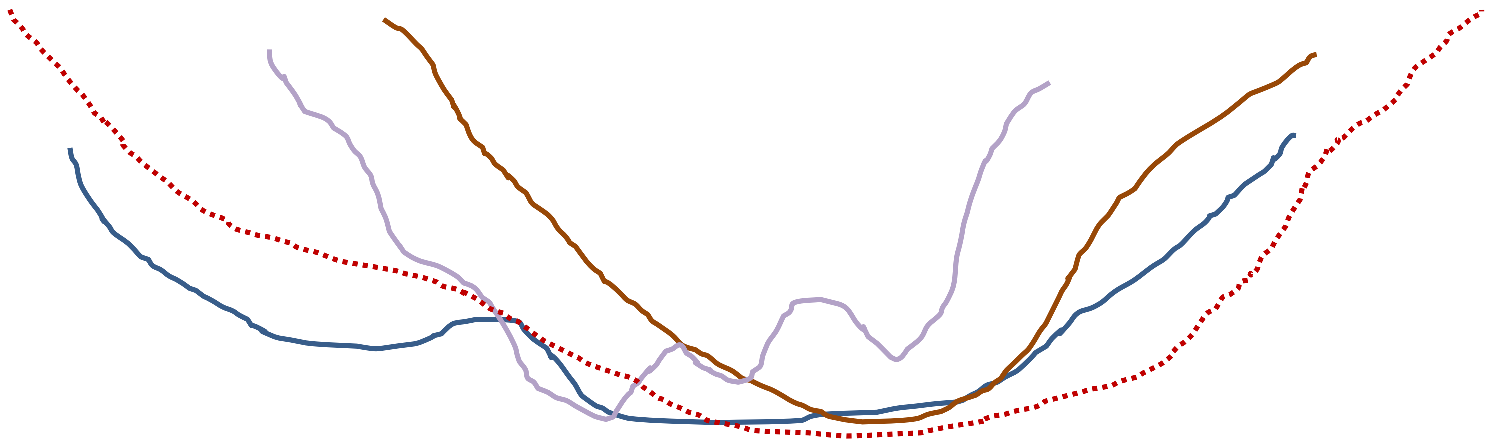
How many samples to choose for each update?

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \, \nabla_{\boldsymbol{w}} \left(\frac{1}{N} \sum_{i=1}^{N} l_i(\boldsymbol{w}) \right)$$

- Vanilla gradient descent: use samples in a fixed sequence: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \, \nabla_{\boldsymbol{w}} l_i(\boldsymbol{w})$
- Stochastic gradient descent: Choose a random sample for each update. In practice, we make random mini-batches based on computational resources
  - Divide N samples into equal sized batches
  - Update weights once per batch
  - One epoch completes when all samples used once
- Batch gradient descent: use all samples, and update once per epoch for all samples

Loss of different sets of samples

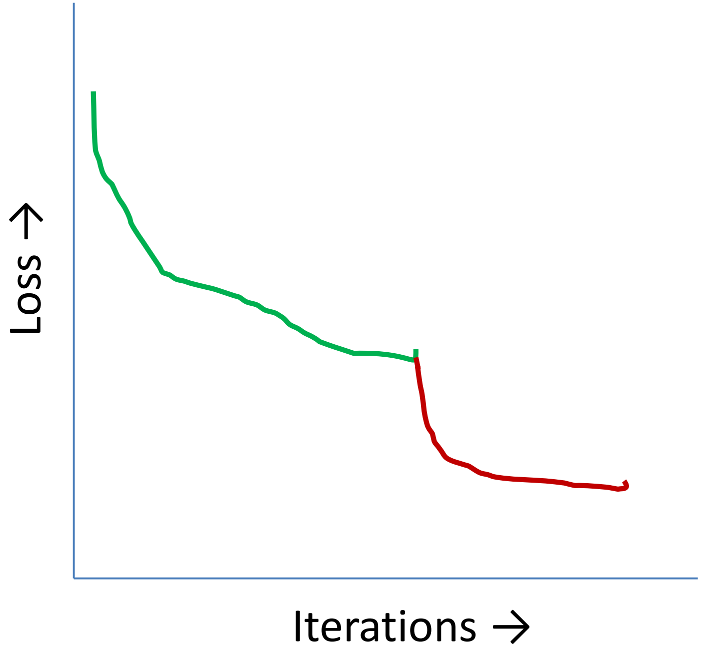
- Different mini-batches (or samples) have their own loss surfaces
- The loss surface of the entire training sample (dotted) may be different
- Local minima of one loss surface may not be local minima of another one
- This helps us escape local minima using stochastic or batch gradient descent
- Mini-batch size depends on computational resource utilization

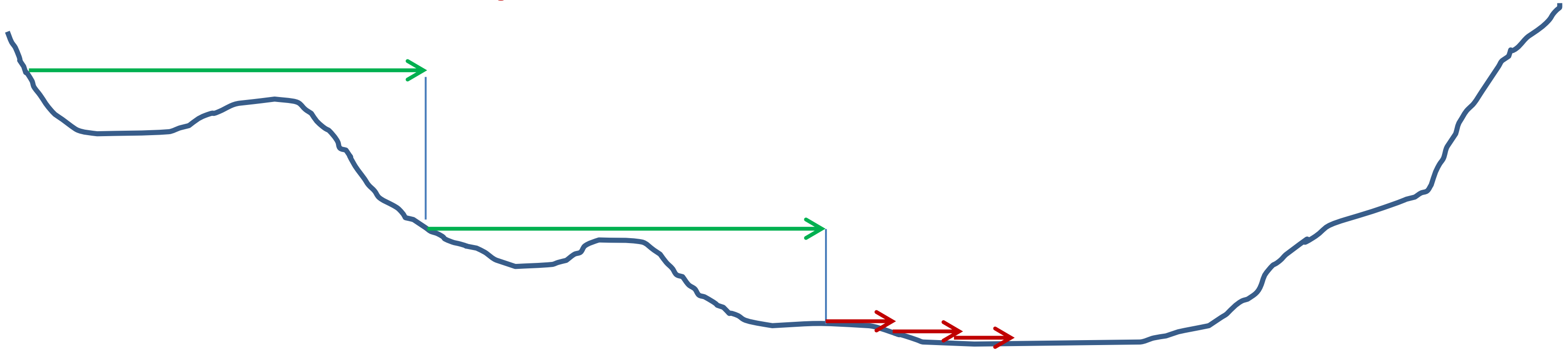


Contents

Learning rate decay

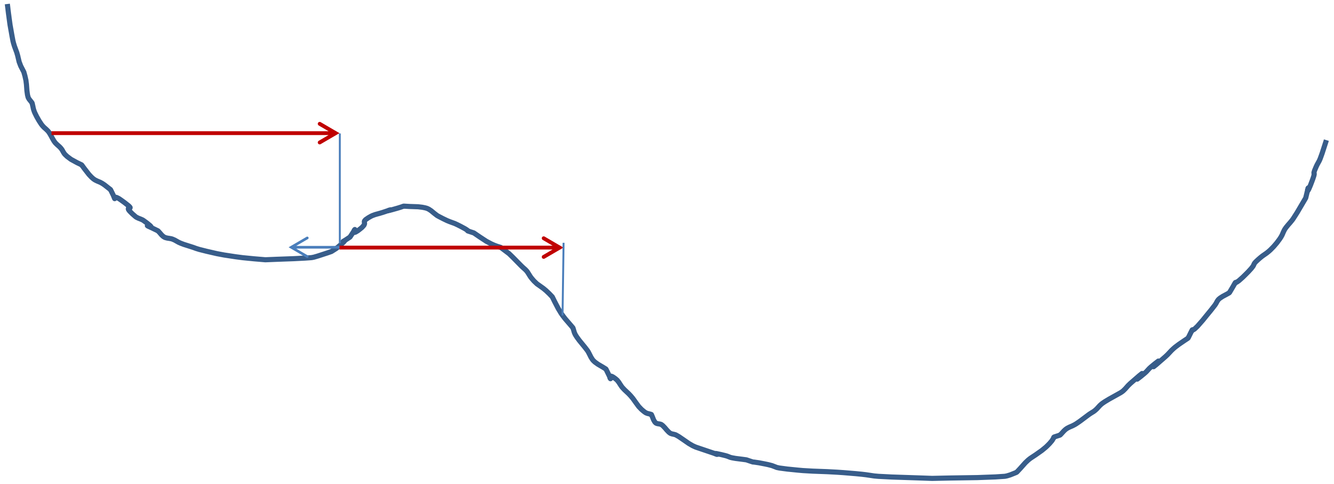
- Initially use large learning rate: further from the global minima, we need to rapidly converge
- Later, reduce the learning rate: Closer to the solution we need to start fine-tuning with smaller steps





Momentum

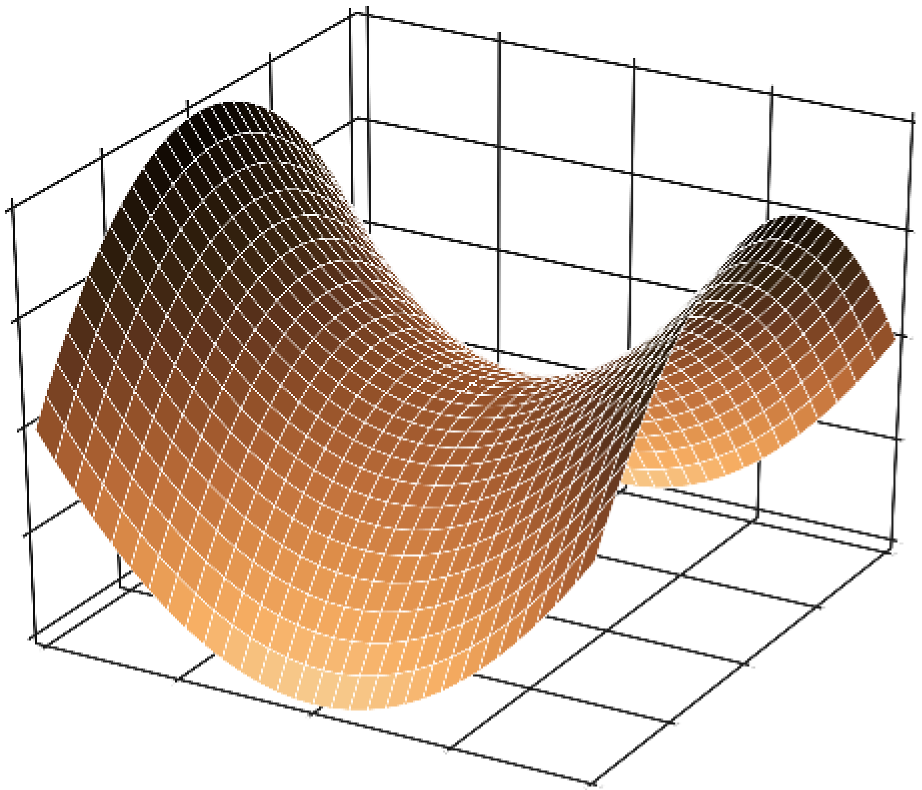
- Momentum means using the memory of previous step to build up speed or to slow down with forgetting factor α; $0 \le \alpha < 1$



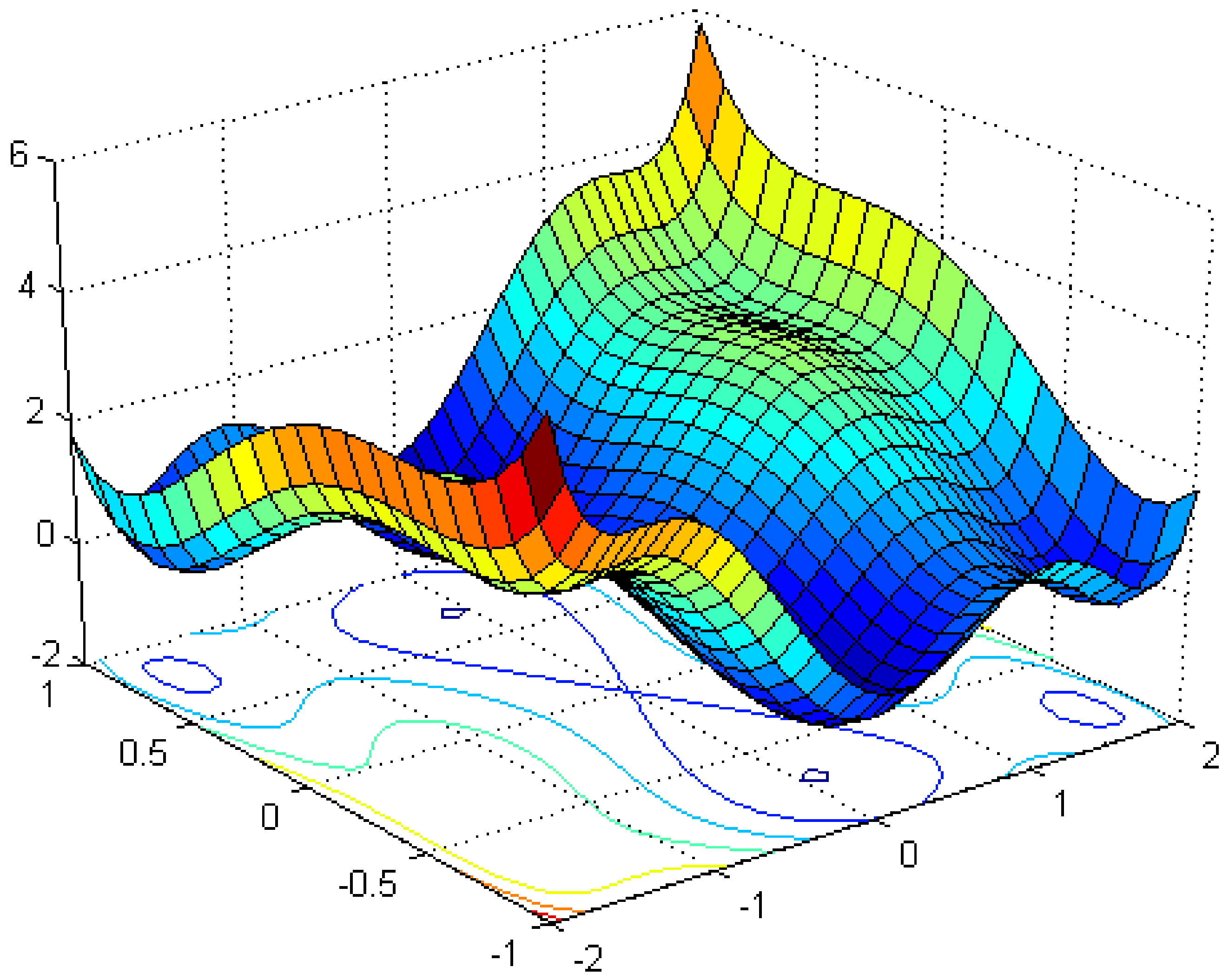
$$\Delta \boldsymbol{w}^{(t)} = \alpha\, \Delta \boldsymbol{w}^{(t-1)} - \eta\, \nabla_{\boldsymbol{w}} L(f_{\boldsymbol{w}}(\boldsymbol{X}), \boldsymbol{y}))$$

Saddle points, Hessian and long local furrows

- In multiple dimensions:
  - Some weights may have reached a local minima while others have not
  - Some weights may have almost zero gradient



Complicated loss functions



Saddle points, Hessian and long local furrows

- This all is characterized by second derivative matrix called the Hessian, which is difficult to compute and invert
- Different techniques try to approximate Hessian computation and inversion
- Usually, they treat each weight independently and set learning rates for each differently
- Examples: RMSprop, ADAM, Nestrov momentum