Neural Network

Perceptron, Perceptron Learning Algorithm, Multilayer Perceptron, Back-propagation, Hyper-parameter tuning, Optimizer, and Regularization

Dr. Rajesh Kumar Mundotiya

Perceptron — 2^{2^n} $n \rightarrow$ # of inputs (2) $\Rightarrow$ $2^{2^2} \Rightarrow 2^4 = 16$

- Not support non-linear
- Consider equal weight to the inputs
- Boolean weights

$$y = 1 \quad \text{if} \sum_{i=0}^{n} w_i x_i \geq 0 *$$

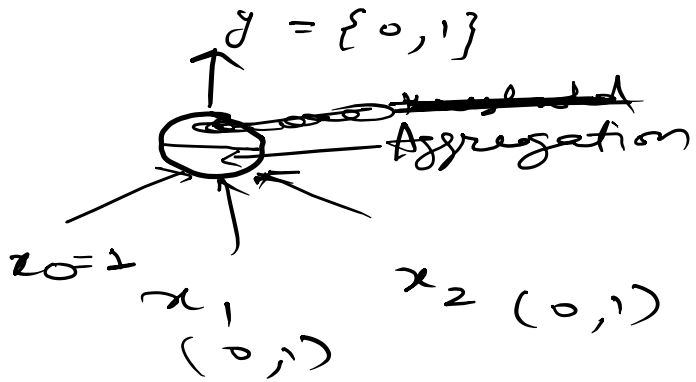
$$= 0 \quad \text{if} \sum_{i=0}^{n} w_i x_i < 0$$

$$y = 1 \quad \text{if} \sum_{i=1}^{n} w_i x_i \geq \theta$$

$$y = 0 \quad \text{if} \sum_{i=1}^{n} w_i x_i < \theta$$

$y = 1 \rightarrow \sum_{i=1}^{n} w_i x_i - \theta \geq 0$

$0 \rightarrow < 0$



$w_0 = -\theta$

$x_0 = 1$

$$\sum_{i=1}^{n} w_i x_i + w_0 x_0 \geq 0$$

* $y = 1 \quad \sum_{i=0}^{n} w_i x_i \geq 0$

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

$\left(\omega_0 + \sum_{i=1}^{n} \omega_i x_i\right) < 0$

$\omega_0 + \text{''} \geq 0$

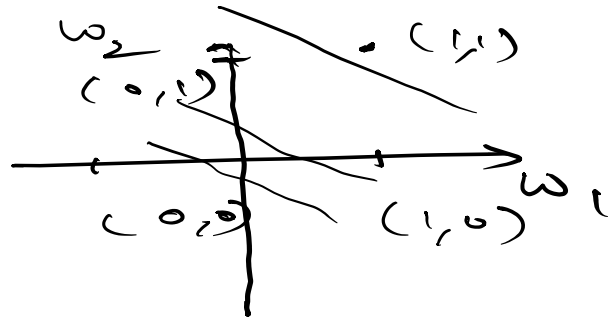
$\omega_0 + \text{''} \geq 0$

$\omega_0 + \text{''} \geq 0$

$\omega_0 = -1, \ \omega_1 = +1.1, \ \omega_2 = 1.1$

$\omega_0 = -1, \ \omega_1 = 0.45$

$\omega_2 = 0.45$



$-1 + 1.1 \times 0 + 1.1 \times 0 \leq 0$

Perceptron learning Algorithm -

$P \leftarrow A_1$; $N \leftarrow L_0$

Initialize $\omega = [\omega_1, \omega_2, \ldots \omega_n]$

while ! converge do

{

random $x \in P \cup N$

if $x \in P$ and $\sum \omega_i x_i < 0$

$\omega = \omega + x$

if $x \in N$ and $\sum \omega_i x_i \geq 0$ $\omega = \omega - x$

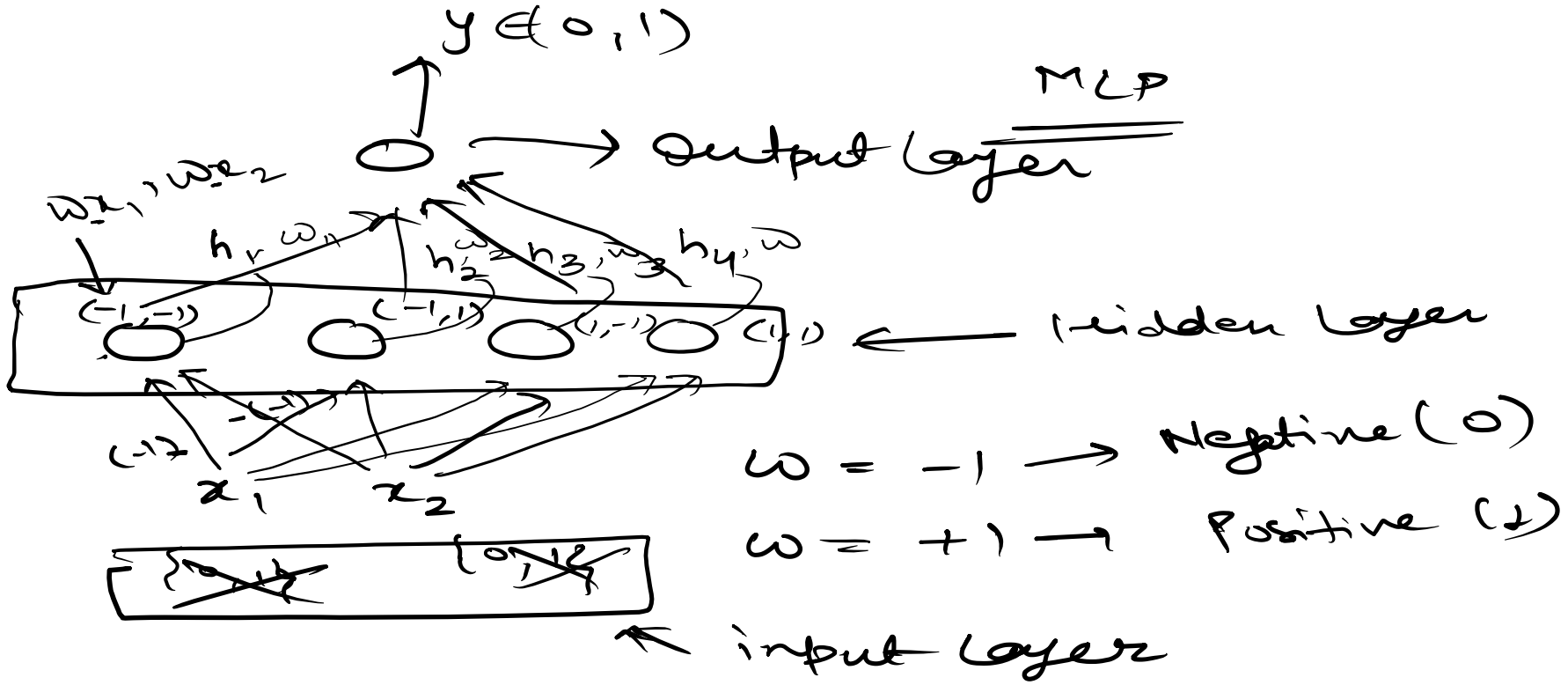
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \Rightarrow w_0 < 0$

$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \Rightarrow w_2 \geq -w_0$

$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \Rightarrow w_1 \geq -w_0$ } ← contrast

$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \Rightarrow w_1 + w_2 < -w_0$ ←



$w = -1 \rightarrow$ Negative (0)

$w = +1 \rightarrow$ Positive (1)

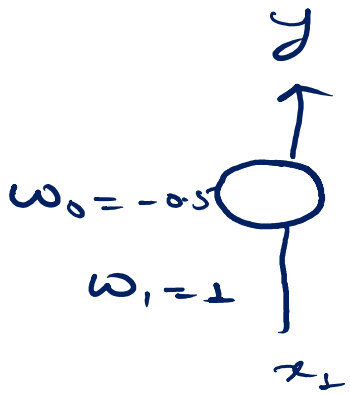
x_1	x_2	XOR	h_1	h_2	h_3	h_4	$\sum_{i=1}^{4} w_i x_i \geq 0$
0	0	0	1	0	0	0	w_1
0	1	1	0	1	0	0	w_2
1	0	1	0	0	1	0	w_3
1	1	0	0	0	0	1	w_4

$f(x)$ (Sigmoid) = Step function

$$\frac{1}{1+e^{-x}} \Rightarrow \frac{1}{1+e^{-(w^T x)}}$$

$w^T x \to \infty \Rightarrow 1$ $\quad w^T x = 0 \Rightarrow 0.5$?

$w^T x \to -\infty \Rightarrow 0$



$y = w_1 x_1 + (-0.5) \Rightarrow x_1 \to 0.5 \Rightarrow$

$(1) \; -0.5 \Rightarrow 0.5$

$(0) \; -0.5 \Rightarrow -0.5$

$0.5 - 0.5 \Rightarrow 0$

$0.51 - 0.5 \Rightarrow 0.01$

$0.01 - 0.5 \Rightarrow -0.49$

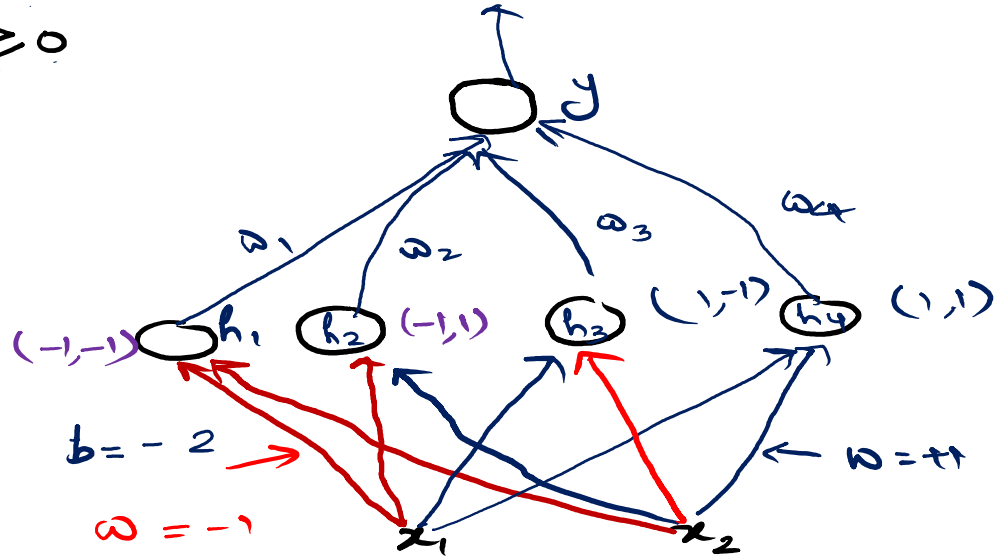
$-0.01 - 0.5 \Rightarrow -0.51$

$f(x)$

$L = (x_{Act} - x_{pred.})$

$$\frac{\partial L}{\partial w} = \frac{\partial (y - f(x))}{\partial w} \Rightarrow \Delta w$$

$w = w + x$ $\quad$ $w_{new} = w_{old} - \Delta w$



Backpropagation

- Every layer except the output layer includes a bias neuron and is fully connected to the next layer.
- When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN).
- *Feedforward neural network* - signal flows only in one direction
- *Backpropagation* – can find out how each connection weight and each bias term should be tweaked in order to reduce the error.
- Backpropagation algorithm is able to compute the gradient (called *automatic differentiation*) of the network's error with regard to every single model parameter.
- Once it has these gradients, it just performs a regular Gradient Descent step.
- Computing gradient and Gradient Descent processes are repeated until the network converges to the solution.

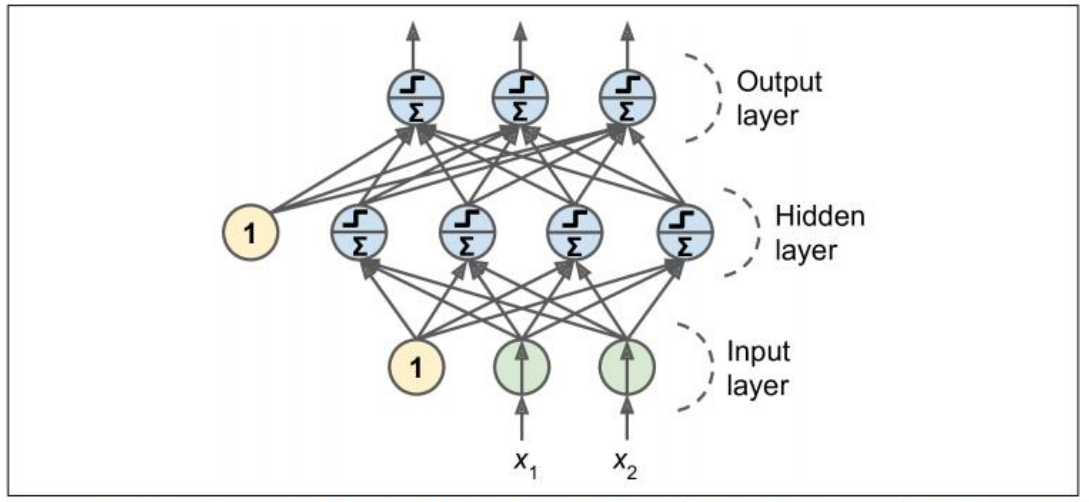


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

autodiff

feedforward + Backpropagate

$\sum w_i x_i$ GD

Gradient

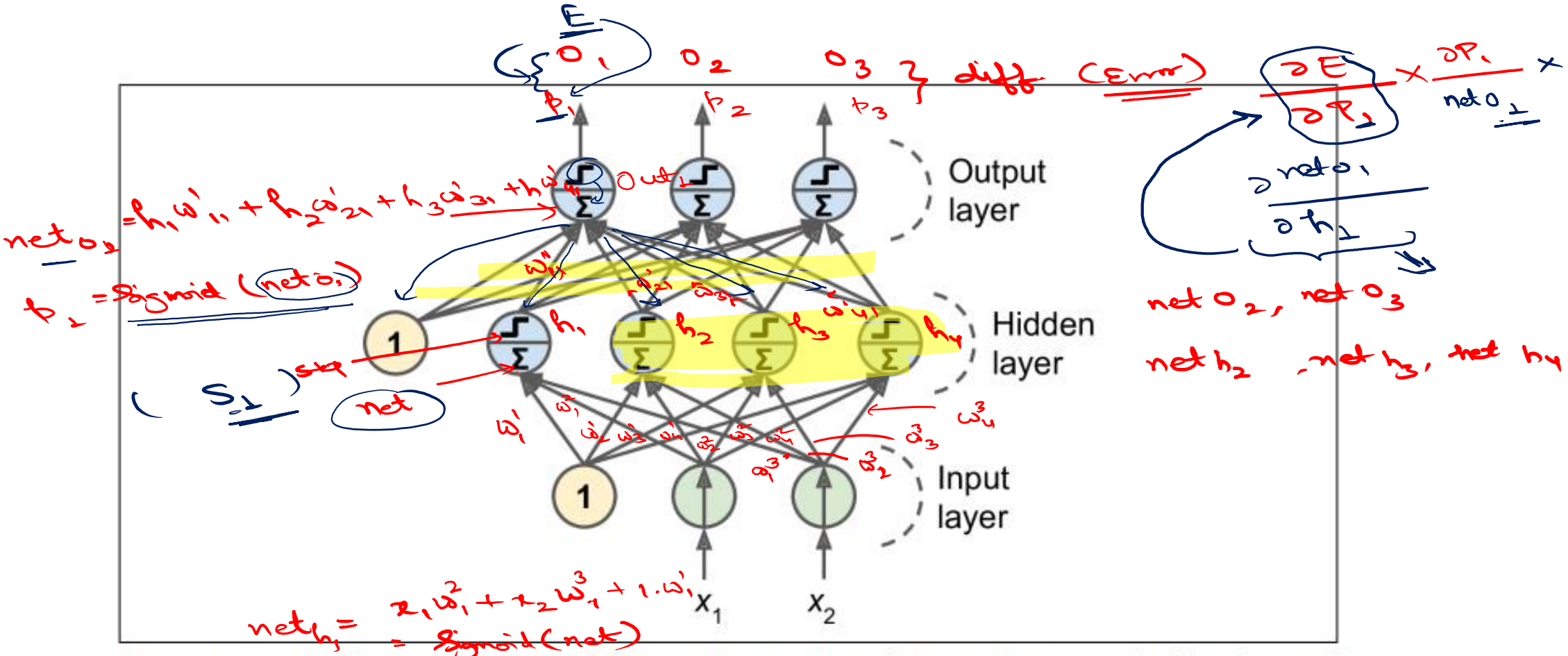


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

For each training instance, the backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).

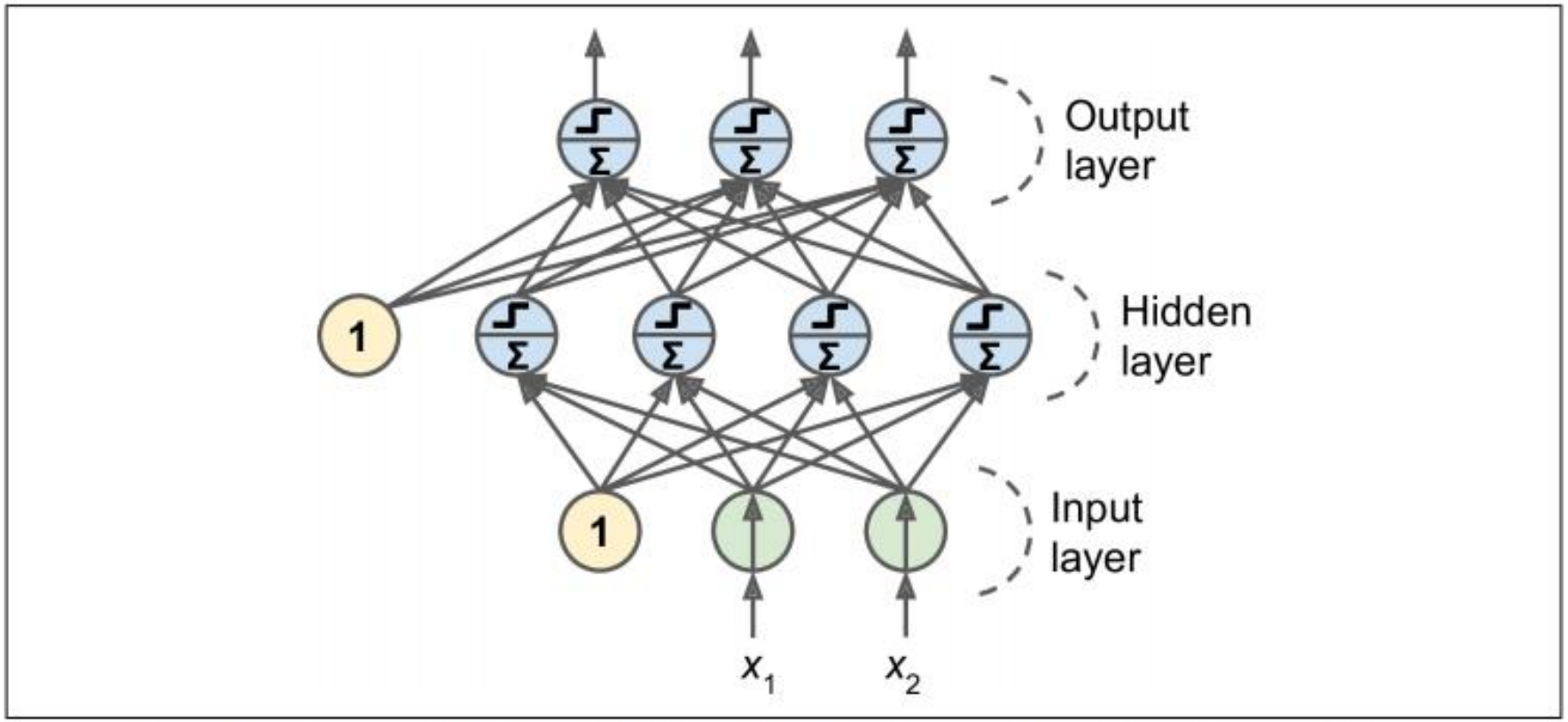


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

- Initialize all the hidden layers' connection weights randomly, or else training will fail.
- For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus, backpropagation will affect them in exactly the same way, so they will remain identical.

Regression and Classification MLPs

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

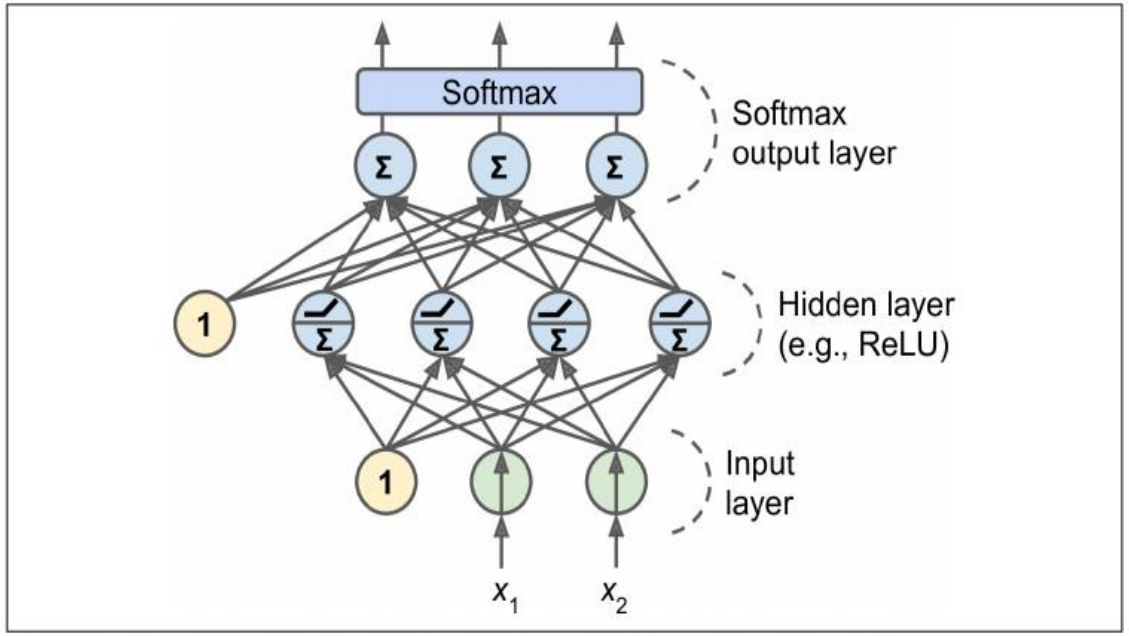


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Neural Network Hyperparameters

- There are many hyperparameters to tweak.
  - Network architecture
  - Number of layers
  - Number of neurons per layer
  - Type of activation function to use in each layer
  - Weight initialization logic, and much more.
- How do you know what combination of hyperparameters is the best for your task?
  - try many hyperparameter combinations and see which works best on the validation set (or use K-fold cross-validation).

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

- ## Number of Hidden Layers
  - Real-world data is often structured in a hierarchical way (such as a forest).
  - Deep neural networks automatically take advantage of:
    - lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations)
    - intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles),
    - highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).
  - Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets.

- ## Number of Neurons per Hidden Layer
  - MNIST task requires $28 \times 28 = 784$ input neurons and 10 output neurons.
  - For the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer (concept of “stretch pants”)
  - Even the same number of neurons in all hidden layers sometimes performs better

- Type of activation function-
  - Compute the gradient of the cost function with regard to each parameter in the network
    - Using gradients to update each parameter with a Gradient Descent step.
- Vanishing gradients problem- gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
  - Gradient Descent update leaves the lower layers' connection weights virtually unchanged
    - training never converges to a good solution.

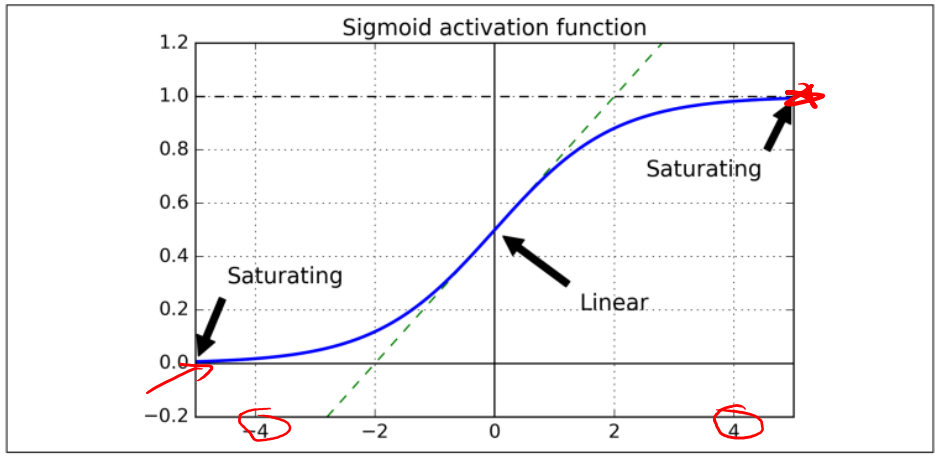


Figure 11-1. Logistic activation function saturation

- Exploding gradients problem- gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges.

- Weight Initialization

- Xavier initialization or Glorot initialization
  - we need the variance of the outputs of each layer to be equal to the variance of its inputs
  - we need the gradients to have equal variance before and after flowing through a layer in the reverse direction
  - *fan-in* and *fan-out* are the input and output neurons, respectively, of the layer and *fan*avg = (*fan*in+ *fan*out)/2
  - connection weights of each layer must be initialized randomly

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

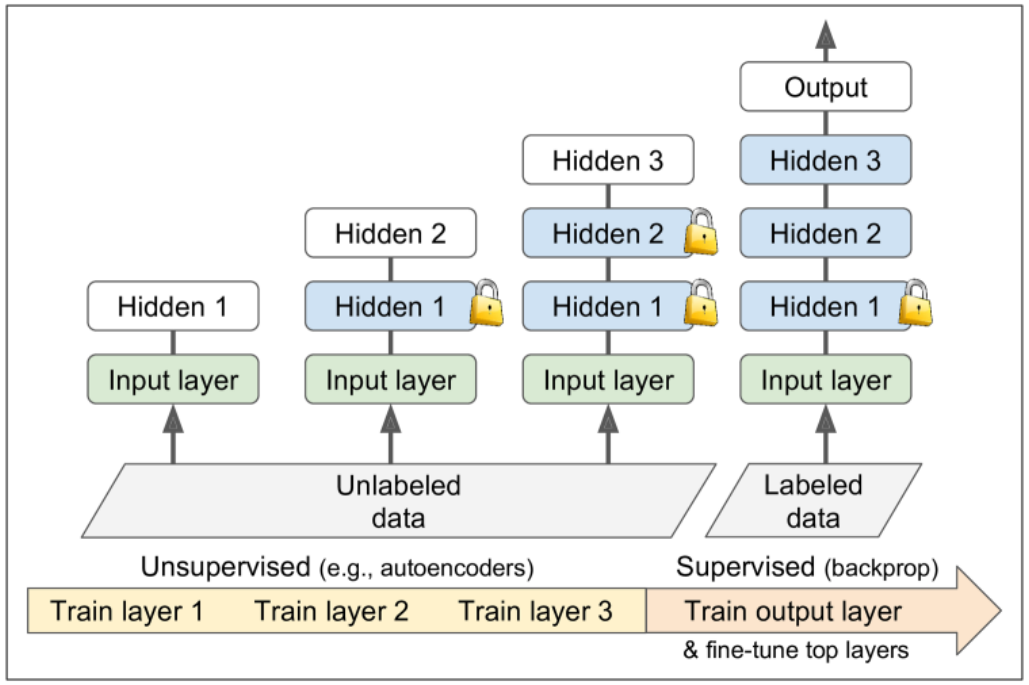
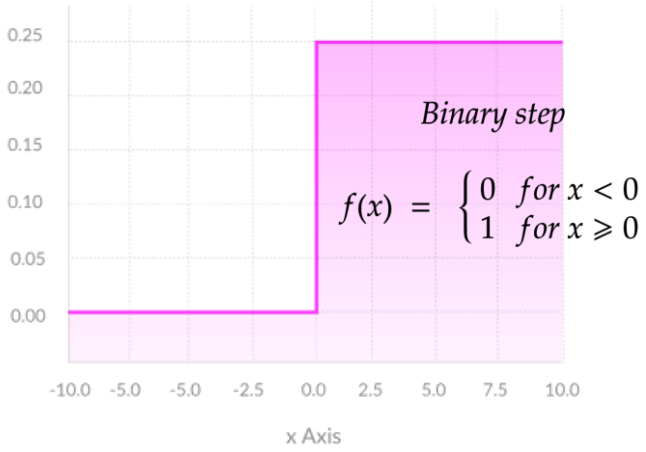


Figure 11-5. In unsupervised training, a model is trained on the unlabeled data (or on all the data) using an unsupervised learning technique, then it is fine-tuned for the final task on the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

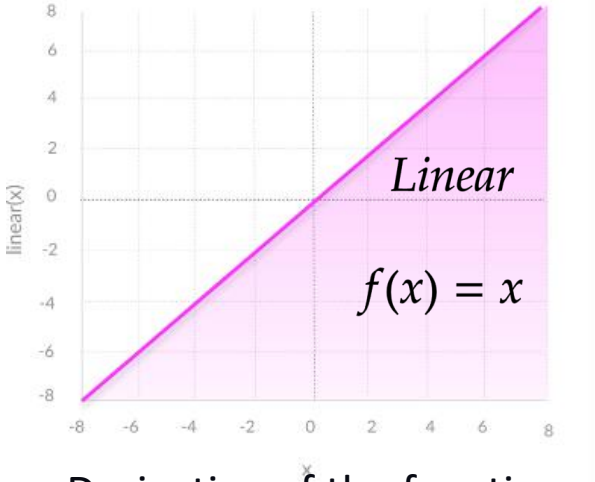
- Unsupervised Pretraining
  - Greedy layer-wise pretraining
    - First train an unsupervised model with a single layer,
      - then freeze that layer and add another one on top of it,
    - train the model again (effectively just training the new layer),
      - then freeze the new layer and add another layer on top of it,
    - train the model again, and so on.

Zero Centric

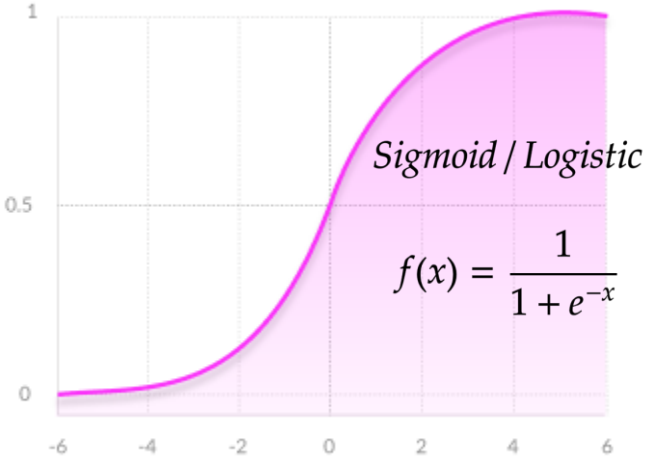
Activation Function



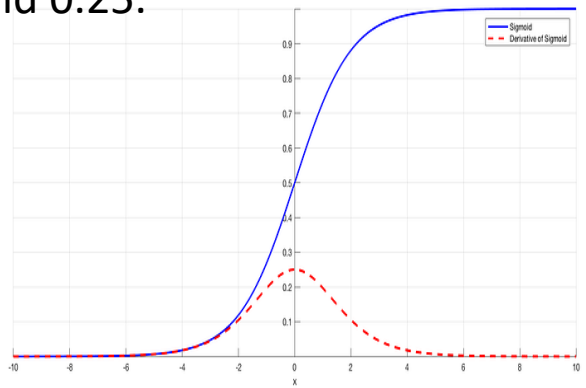
Gradient of the step function is zero, this indicates that there will be no change in the weights during backward propagation.



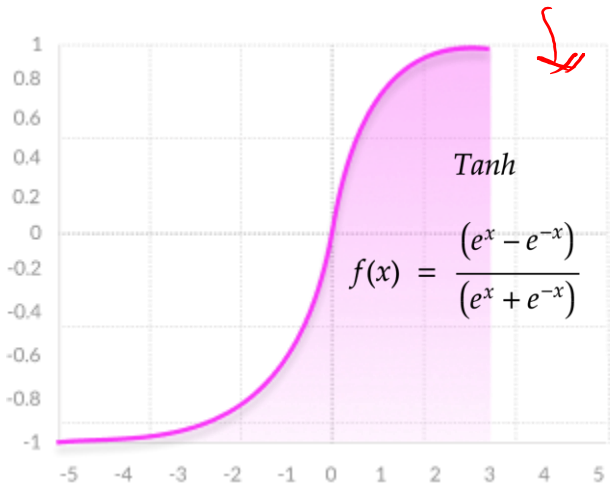
Derivative of the function is a constant and has no relation to the input x.

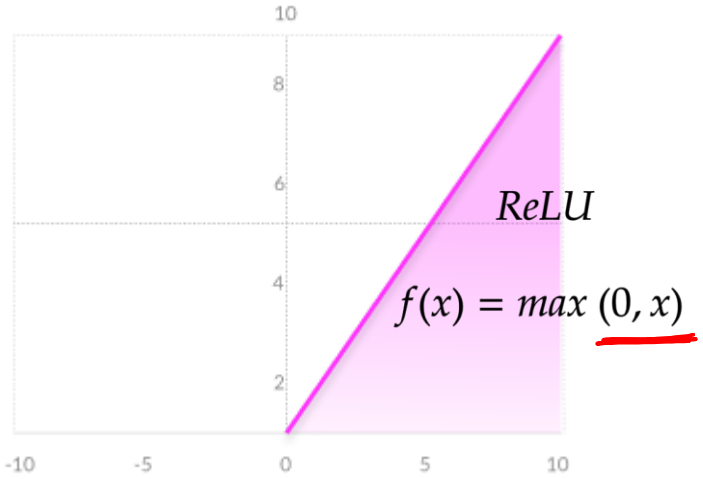


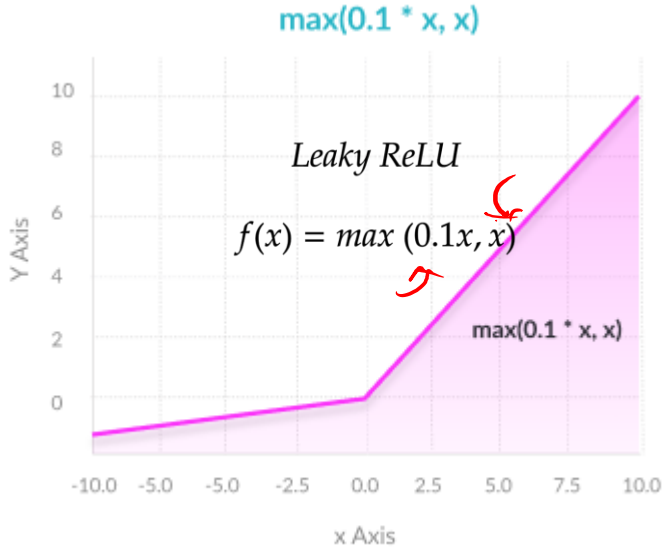
- The function is differentiable and provides a smooth gradient.
- sigmoid function varies between 0 and 0.25.



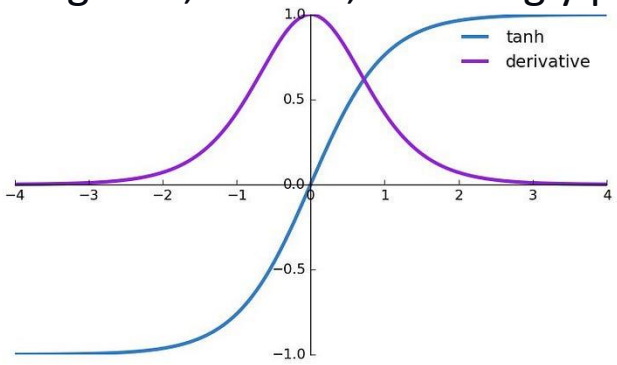
Computationally Expensive



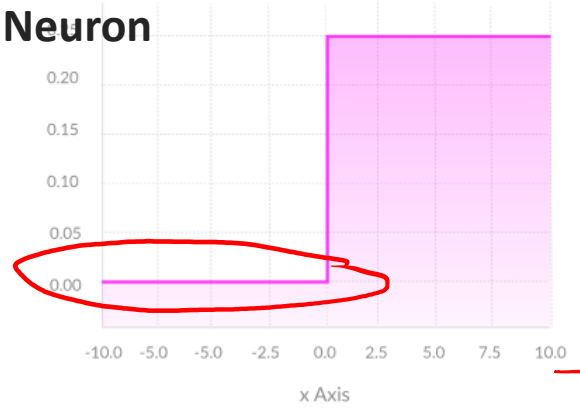


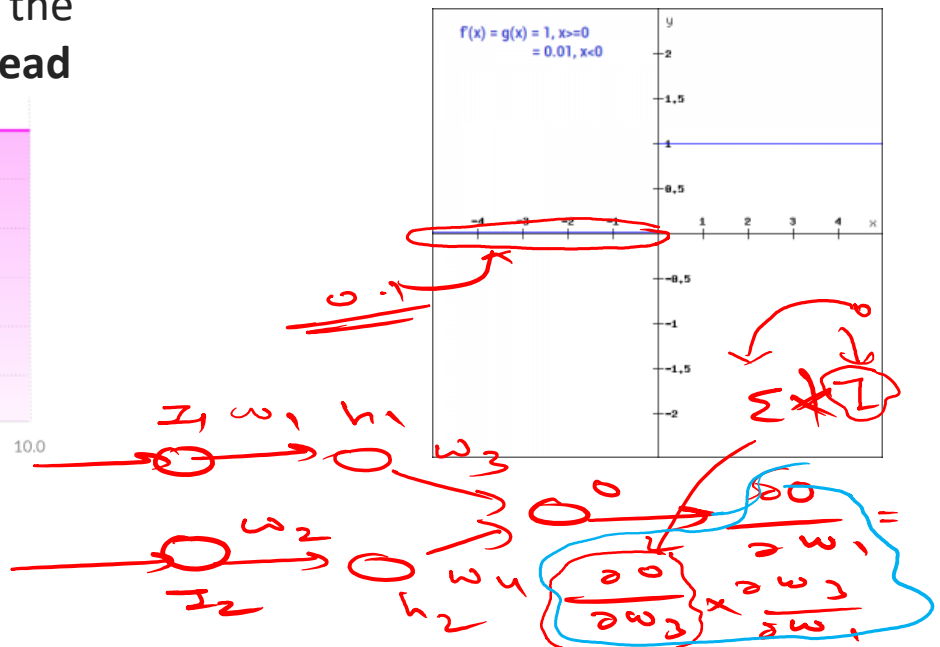


The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.



The result is zero for the negative input values, which means the neuron is not activated - **Dead Neuron**





Parametric ReLU

$f(x) = max(ax, x)$

Softmax

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

GELU

$$f(x) = xP(X \leq x) = x\Phi(x)$$
$$= 0.5x\left(1 + tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

Swish

$$f(x) = x * sigmoid(x)$$

ELU

$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha\left(e^x - 1\right) & for\ x < 0 \end{cases}$$

SELU

$$f(\alpha, x) = \lambda \begin{cases} \alpha\left(e^x - 1\right) & for\ x < 0 \\ x & for\ x \geqslant 0 \end{cases}$$

- Some other activation functions:
  - Parametric ReLU Function
  - Exponential Linear Units (ELUs) Function
  - Softmax Function
  - Swish
  - Gaussian Error Linear Unit (GELU)
  - Scaled Exponential Linear Unit (SELU)

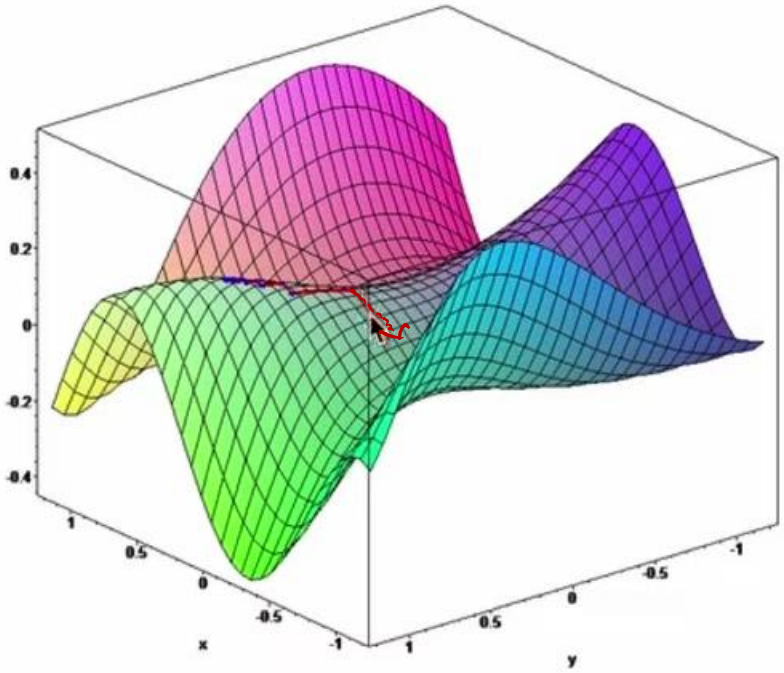
- Gradient Clipping
  - Mitigate the exploding gradients problem by clip the gradients during backpropagation so that they never exceed some threshold.

$C \approx L$

k-Value clip
$[-1, +1]$

$$\text{Gradient vector} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \cdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix} \rightarrow \begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.9 \\ 1.0 \\ 1.0 \\ -1.0 \\ 0.3 \end{bmatrix}$$

Clip back every component of the gradient vector to be in the threshold.

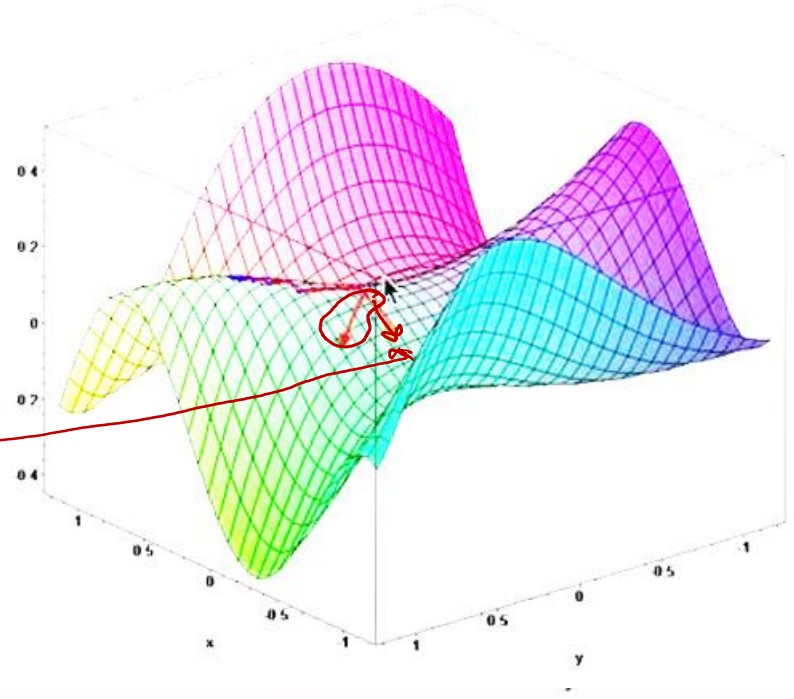


- Gradient Clipping
  - Mitigate the exploding gradients problem by clip the gradients during backpropagation so that they never exceed some threshold.

$$\text{Gradient vector} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \cdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \quad \begin{bmatrix} 0.9 \\ 1.0 \\ 1.0 \\ -1.0 \\ 0.3 \end{bmatrix}$$

Clip back every component of the gradient vector to be in the threshold.



- Gradient Clipping
  - Mitigate the exploding gradients problem by clip the gradients during backpropagation so that they never exceed some threshold.

L_2 - norm

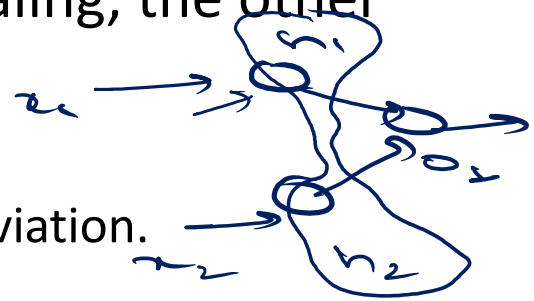
$$\text{Gradient vector} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \cdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.006 \\ 0.021 \\ 1.0 \\ -0.014 \\ 0.002 \end{bmatrix}$$

Clip back every component of the gradient vector to be in the threshold.

"clip by norm" to keep the same direction of the gradient.

- Batch Normalization (BN)
  - He initialization along with ELU (or any variant of ReLU) reduces the danger of the vanishing/exploding gradients problems at the beginning of training.
    - Doesn't guarantee that they won't come back during training.
  - simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
  - In order to zero-center and normalize the inputs
    - the algorithm needs to estimate each input's mean and standard deviation.



Equation 11-3. Batch Normalization algorithm

1. $\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$
4. $\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$

- $\boldsymbol{\mu_B}$ *and* $\boldsymbol{\sigma_B}$ are the vectors of input means and standard deviations respectively, evaluated over the mini-batch B (it contains one mean and standard deviation per input).
- $\boldsymbol{m_B}$ is the number of instances in the mini-batch.
- $\hat{\boldsymbol{x}}^{\boldsymbol{i}}$ is the vector of zero-centered and normalized inputs for instance i.
- $\boldsymbol{\gamma}$ output scale parameter vector for the layer.
- $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer.
- $\boldsymbol{\varepsilon}$ is a smoothing term.
- $\boldsymbol{z}^{\boldsymbol{i}}$ is the output of the BN operation

- During training, BN standardizes its inputs, then rescales and offsets them.
- During testing
  - Sol-1: make predictions for individual instances rather than for batches of instances.
  - Sol-2: wait until the end of training, then run the whole training set through the neural network
    - compute the mean and standard deviation of each input of the BN layer.
    - These “final” input means and standard deviations are used when making predictions.
  - Sol-3: Batch Normalization estimates these ($\gamma, \beta, \sigma, \mu$) final statistics during training by using a moving average of the layer’s input means and standard deviations.

- Optimizer
  - Speed up training
  - Problems with GD for – Learning rate, Saddle point, Local minima
  - Learning Rate Scheduling
    - much too high, training may diverge
    - too low, training will eventually converge

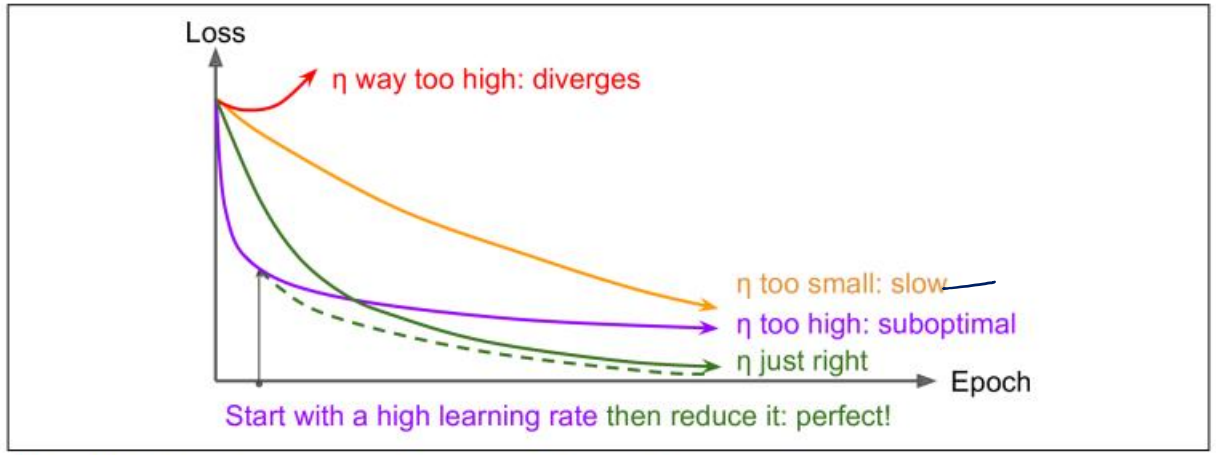
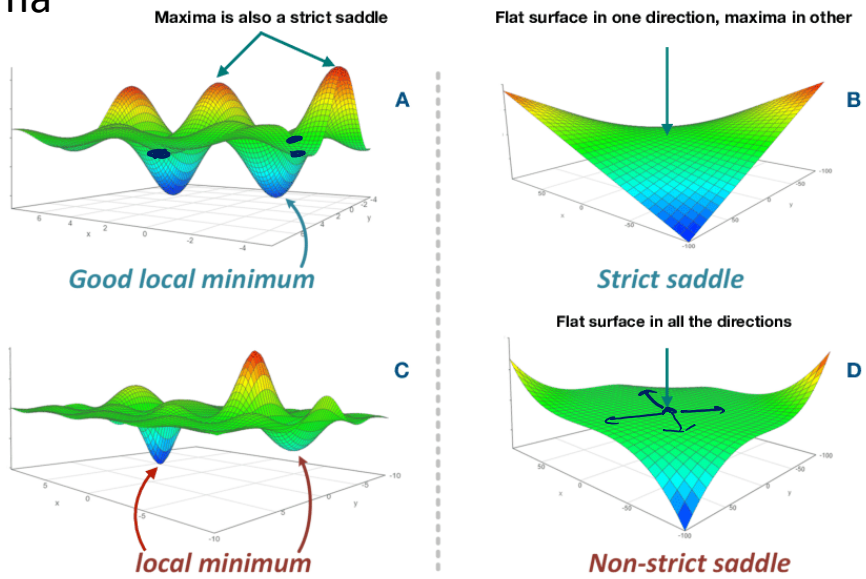


Figure 11-8. Learning curves for various learning rates η



Power scheduling: Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0 / \left(1 + \frac{t}{s}\right)^c$

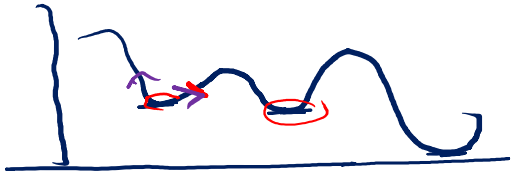
Exponential scheduling: Set the learning rate to $\eta(t) = \eta_0\, 0.1^{\frac{t}{s}}$.

Piecewise constant scheduling: constant learning rate for a number of epochs (e.g., η0 = 0.1 for 5 epochs), then a smaller learning rate for another number of epochs.

Performance scheduling: Measure the validation error every N steps and reduce the learning rate by a factor of λ

m.v

- Local minima
- Saddle point (!)
- High Curvature



SGD with Momentum

if previous gradients are referring to same directions then increase the speed.

$$w_{t+1} = w_t - \eta \Delta w_t$$

$$w_{t+1} = w_t - v_t$$

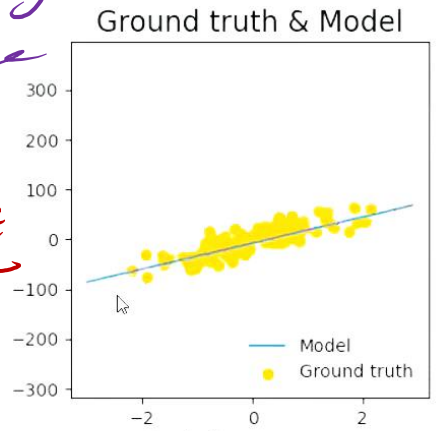
$$v_t = \beta \times v_{t-1} + \eta \Delta w_t$$

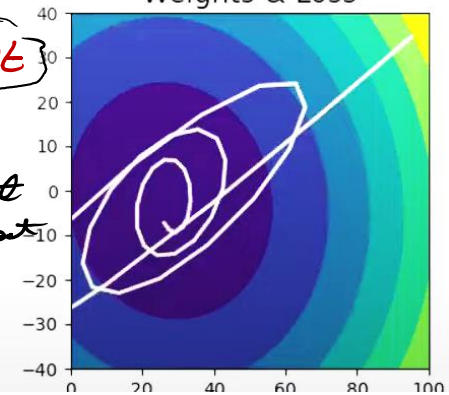
$0 < \beta < 1$ — History of Velocity — $\boxed{0.9}$

$\eta \Delta w_t$ — gradient at that place

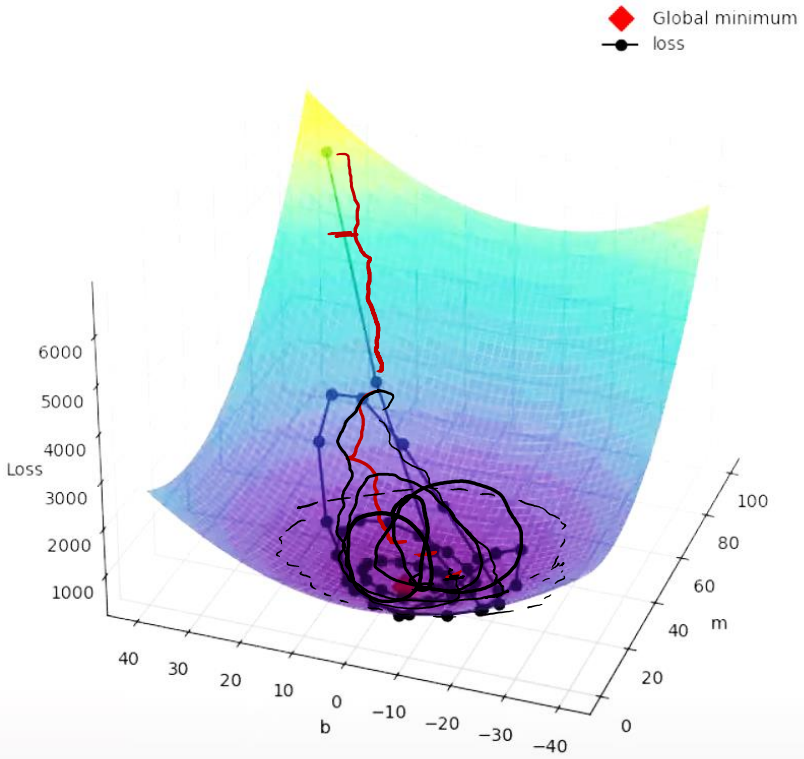
if $\beta = 0 \rightarrow$ GD

$\beta = 1 \rightarrow$





Momentum Optimizer(decay = 0.9)
epoch number: = 47



NAG (Nesterov Accelerate Gradient)

→ first apply velocity

→ then apply the gradient

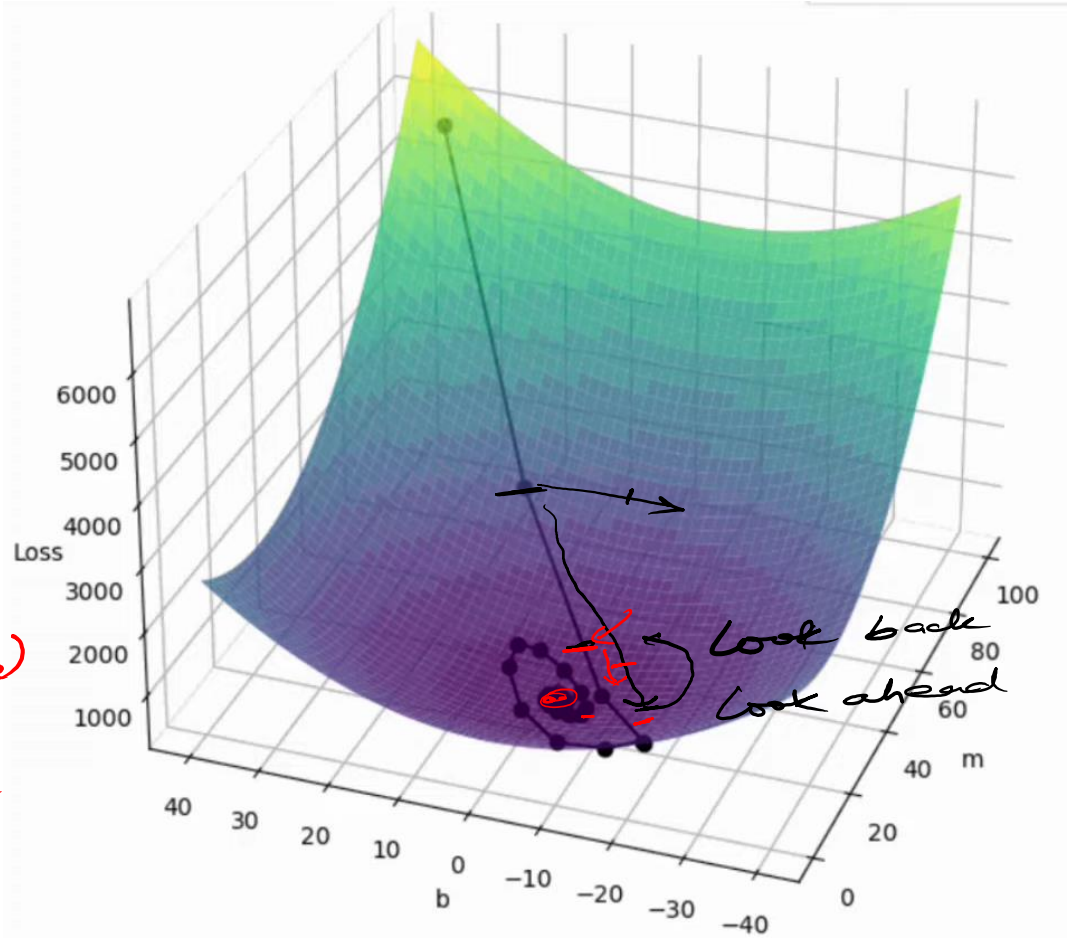
$$\omega_{la} = \omega_t - \beta v_{t-1}$$

$$v_t = \beta v_{t-1} + \eta \Delta \omega_{la}$$

$$\omega_{t+1} = \omega_t - v_t$$

$$v_t = (\omega_t - \omega_{la}) + \eta \Delta \omega_{la}$$

may stuck in local minima



AdaGrad (Adaptive Gradient)

Set LR according to the # of weights -

LR → small → Gradient → big

$$w_{t+1} = w_t - \frac{\eta \Delta w_t}{\sqrt{V_t + \epsilon}}$$

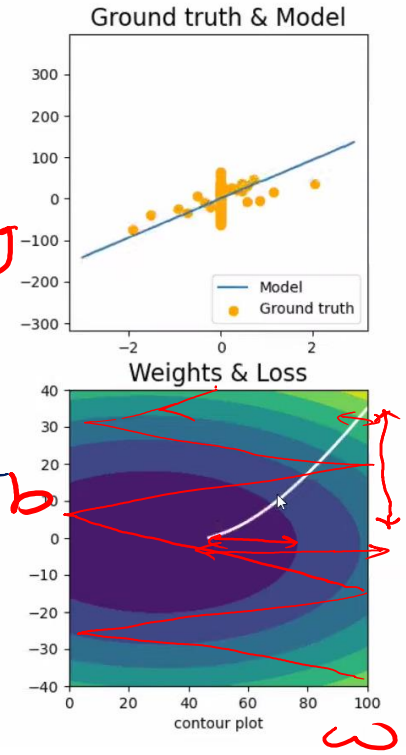
$\sqrt{V_t + \epsilon}$ → small

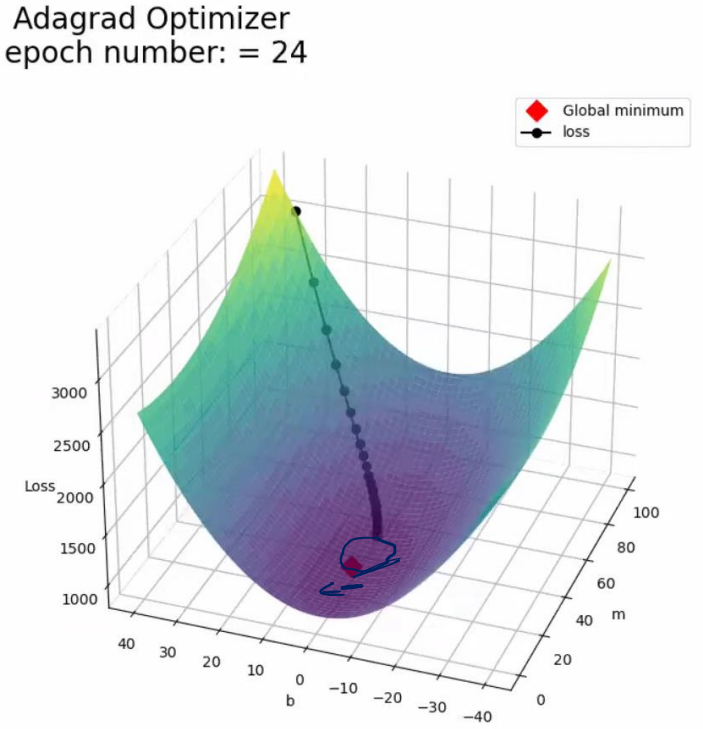
0.01 to 1

$$V_t = V_{t-1} + (\nabla w_t)^2$$

↓ Squared of

Summation over the past gradients





Not preferred in Deep NN.

RMSProp (Root mean square)

$(0 < \beta < 1)$

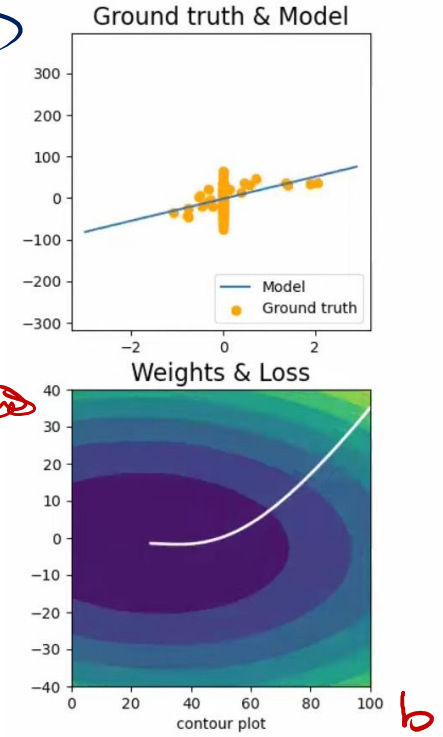
$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$

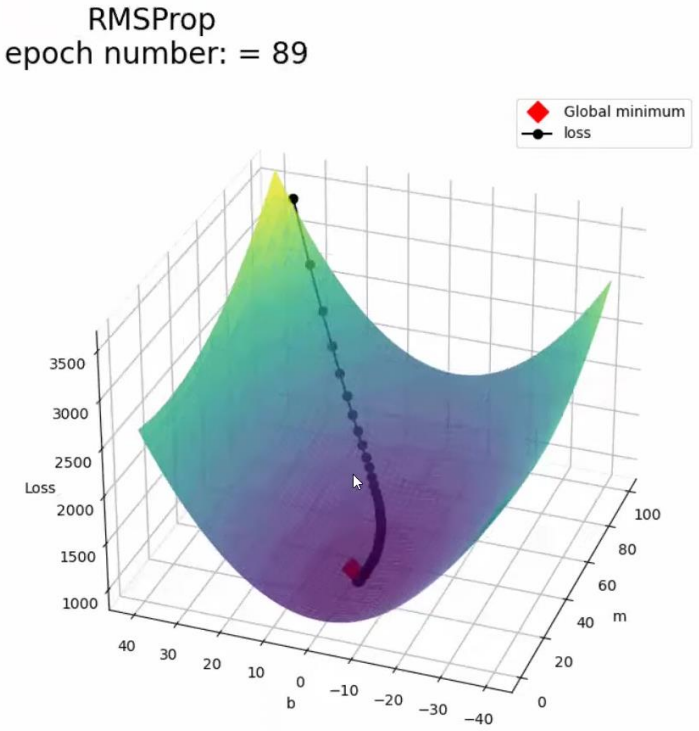
(decay factor)

$$w_{t+1} = w_t - \frac{\eta \nabla w_t}{\sqrt{v_t + \epsilon}}$$

$\beta = 0.9$

To stop v_t get large value by





ADAM

$y = mx + b$

↓

momentum + learning decay

$$w_{t+1} = w_t - \frac{\eta \times m_t}{\sqrt{v_t + \epsilon}}$$

m, b

Same for the bias

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2$$

Regularization

$p = 0.2$

ℓ_1 and ℓ_2 Regularization

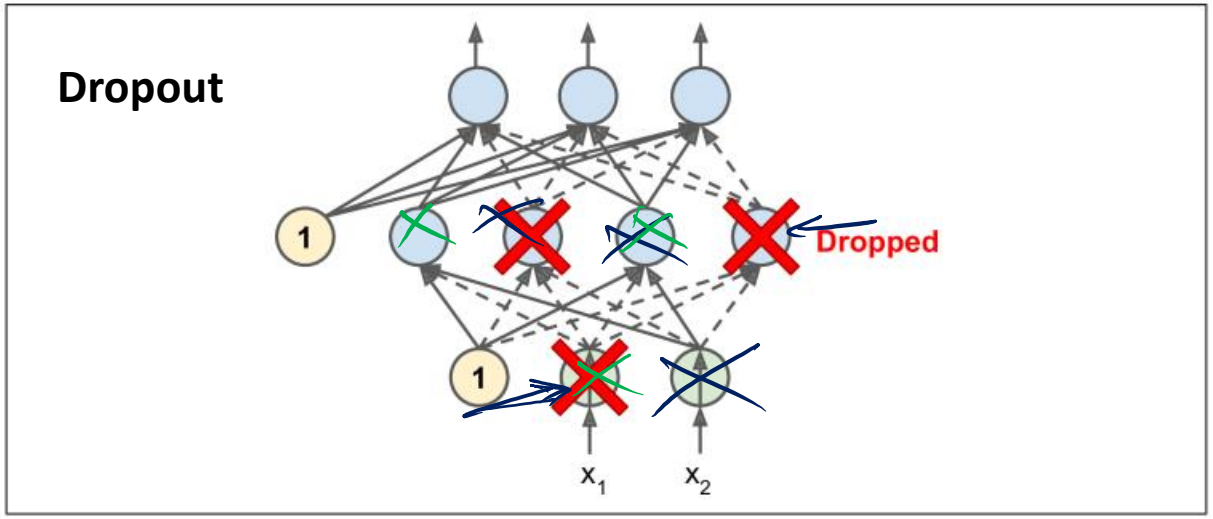


Figure 11-9. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are "dropped out"; these neurons output 0 at this iteration (represented by the dashed arrows)