# Neural networks

Fraida Fund

## Contents

## In this lecture

- Neural network
- Structure of a neural network
- Training a neural network

## From linear to non-linear

### Review: learning non-linear decision boundaries from linear classifiers

- Logistic regression - using basis functions
- SVM - using kernel
- Decision tree - AdaBoost uses multiple linear classifiers (decision stumps)

### Using multiple logistic regressions?

1. Classify into small number of linear regions. Each output from step 1 is a linear classifier with soft decision.
2. Predict class label. Output is weighted average of step 1 weights

### Example: synthetic data

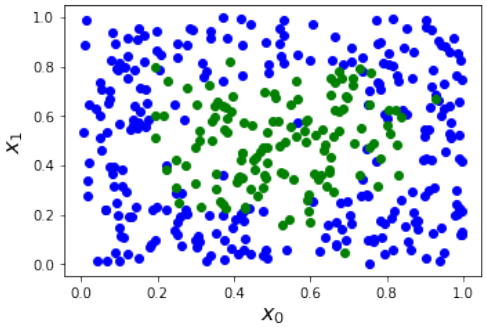



Figure 1: Via Sundeep Rangan

### Model of example two-stage classifier (1)

First step (*hidden layer*):

- Take $N_H = 4$ linear discriminants.

$$\begin{bmatrix} z_{H,1} = w_{H,1}^T x + b_{H,1} \\ \cdots \\ z_{H,N_H} = w_{H,N_H}^T x + b_{H,N_H} \end{bmatrix}$$

- Each makes a soft decision: $u_{H,m} = g(z_{H,m}) = \frac{1}{1+e^{-z_{H,m}}}$

**Model of example two-stage classifier (2)**

Second step (*output layer*):

- Linear discriminant using output of previous stage as features:

$$z_O = w_O^T u_H + b_O$$

- Soft decision:

$$u_O = g(z_O) = \frac{1}{1 + e^{-z_O}}$$

**Example: multiple logistic regressions**

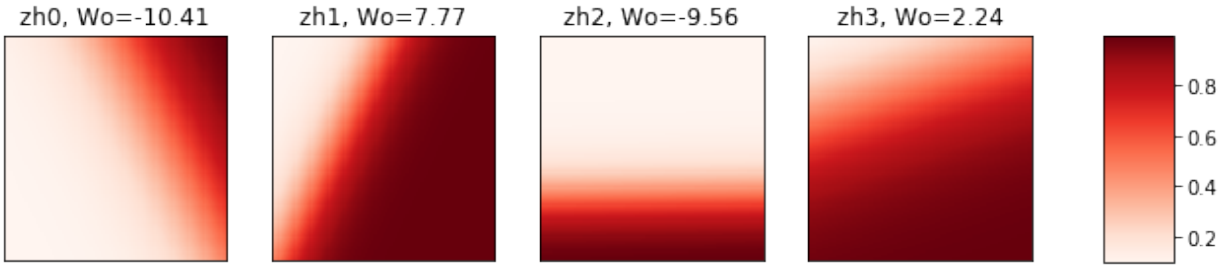



Figure 2: Via Sundeep Rangan

**Example: weighted average output**

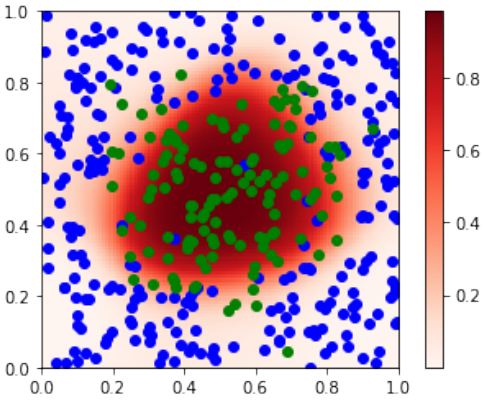



Figure 3: Via Sundeep Rangan

**Matrix form of two stage classifier**

- Hidden layer: $\mathbf{z}_H = \mathbf{W}_H^T \mathbf{x} + \mathbf{b}_H, \quad \mathbf{u}_H = g(\mathbf{z}_H)$
- Output layer: $z_O = \mathbf{W}_O^T \mathbf{u}_H + \mathbf{b}_O, \quad \mathbf{u}_O = g(\mathbf{z}_O)$

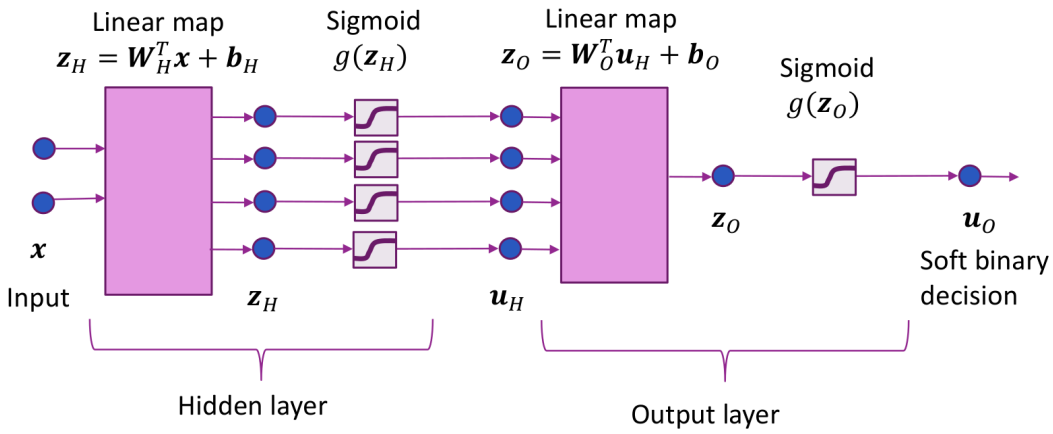



Figure 4: Two-stage classifier. Image credit: Sundeep Rangan

**Illustration of two-stage classifier**

**Training the two-stage classifier**

- From final stage: $z_o = F(\mathbf{x}, \theta)$ where parameters $\theta = (\mathbf{W}_H, \mathbf{W}_o, b_H, b_o)$
- Given training data $(\mathbf{x}_i, y_i), i = 1, \ldots, N$
- Loss function $L(\theta) := -\sum_{i=1}^{N} \ln P(y_i|\mathbf{x}_i, \theta)$
- Choose parameters to minimize loss: $\hat{\theta} = \text{argmin}_\theta \, L(\theta)$

## Neural networks

**Biological inspiration**

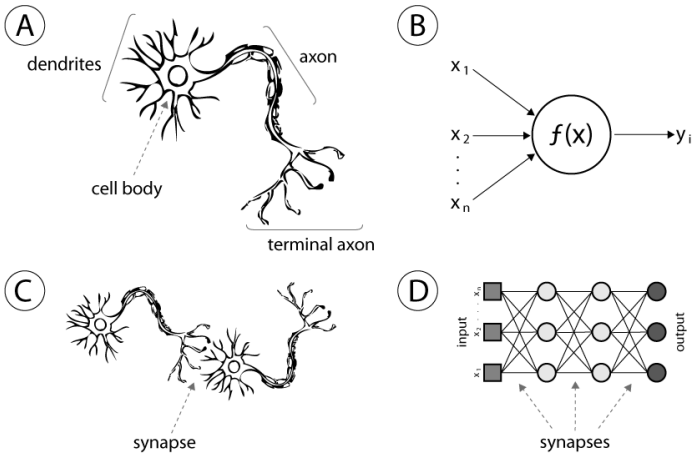



Figure 5: A biological neuron accepts inputs via dendrites, "adds" them together within cell body, and once some electrical potential is reached, "fires" via axons. Synaptic weight (the influence the firing of one neuron has on another) changes over time ("learning"). Image via: http://http://doi.org/10.5772/51275

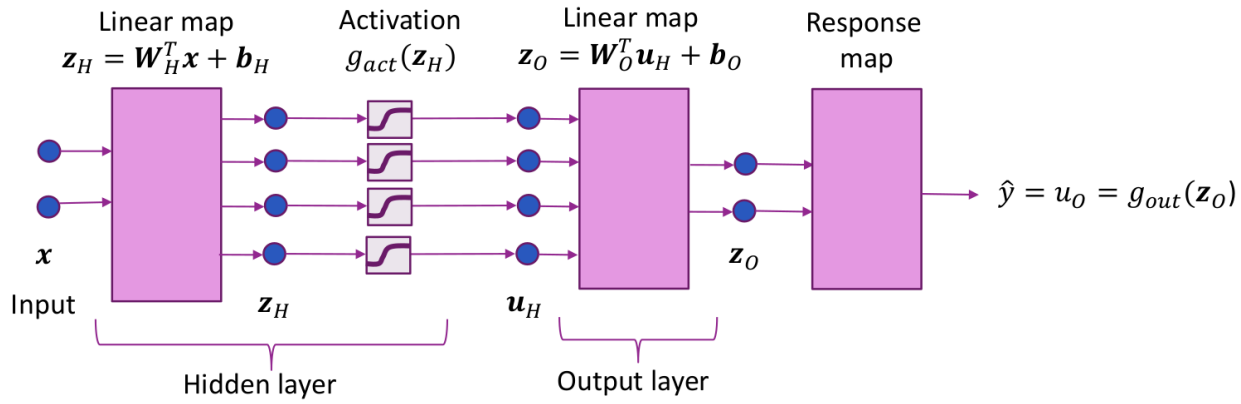



Figure 6: Block diagram. Image credit: Sundeep Rangan

**General structure (illustration)**

**Terminology**

- **Hidden variables**: the variables $\mathbf{z}_H, \mathbf{u}_H$, which are not directly observed.
- **Hidden units**: the functions that compute $z_{H,i}$ and $u_{H,i}$.
- **Activation function**: the function $g(z)$
- **Output units**: the functions that compute $z_{O,i}$.

**General structure: input and hidden layers**

Input: $\mathbf{x} = (x_1, \ldots, x_d)$

Each hidden layer:

- Linear transform: $\mathbf{z}_H = \mathbf{W}_H^T \mathbf{x} + \mathbf{b}_H$
- Activation function $\mathbf{u}_H = g_{act}(\mathbf{z}_H)$

**General structure: output layer**

Output layer:

- Linear transform: $\mathbf{z}_O = \mathbf{W}_O^T \mathbf{u}_H + \mathbf{b}_O$
- Output function $\mathbf{u}_O = g_{out}(\mathbf{z}_O)$

Response map depends on type of response -

**General structure: response map, binary classification**

For binary classification, $y = \pm 1$:

- $z_O$ is scalar
- Hard decision: $\hat{y} = \text{sign}(z_O)$
- Soft decision: $P(y = 1|x) = \frac{1}{1+e^{-z_O}}$

**General structure: response map, classification**

For multi-class classification, $y = 1, \ldots, K$:

- $\mathbf{z}_O = [z_{O,1}, \ldots, z_{O,K}]$ is a vector
- Hard decision: $\hat{y} = \text{argmax}_k \, z_{O,k}$
- Soft decision: $P(y = k|x) = \frac{e^{z_{O,k}}}{\sum_\ell e^{-z_\ell}}$ (softmax)

**General structure: response map, regression**

For regression, $y \in R^K$:

- Linear: $\hat{y} = z_O$

**Activation functions: identity?**

- Suppose we use $g(z) = z$ (identity function) as activation function throughout the network.
- The network can only achieve linear decision boundary!
- To get non-linear decision boundary, need non-linear activation functions.
- Universal approximation theorem: under certain conditions, with enough (finite) hidden nodes, can approximate any continuous real-valued function, to any degree of precision. But only with non-linear decision boundary!

**Activation functions: binary step**

- Not differentiable at $x = 0$, has $0$ derivative everywhere else.
- Not useful for gradient-based optimization methods.

**Activation functions: some choices**

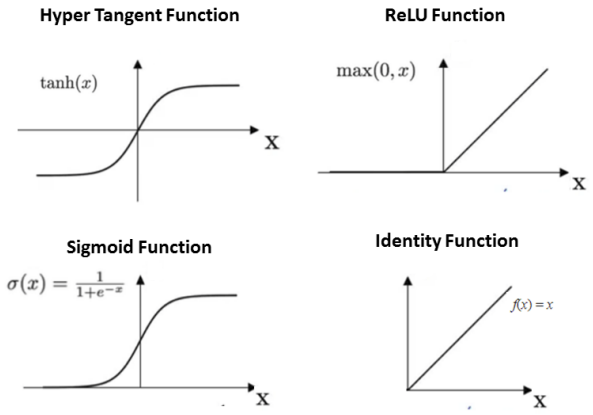



Figure 7: Most common activation functions

**Dimension (1)**

- $N_I$ = input dimension, number of features
- $N_H$ = number of hidden units (tuning parameter?)
- $N_O$ = output dimension, number of classes

**Dimension (2)**

| Parameter | Symbol | Number of parameters |
|---|---|---|
| Hidden layer: bias | $b_H$ | $N_H$ |
| Hidden layer: weights | $W_H$ | $N_H N_I$ |
| Output layer: bias | $b_O$ | $N_O$ |
| Output layer: weights | $W_O$ | $N_O N_H$ |
| Total | | $N_H(N_I+1)+N_O(N_H+1)$ |

## Training a neural network

- Given training data $(\mathbf{x}_i, y_i), i = 1, \ldots, N$
- Parameters to learn are $\theta = (\mathbf{W}_H, \mathbf{W}_o, b_H, b_o)$
- Choose parameters to minimize loss function $L(\theta)$:

$$\hat{\theta} = \underset{\theta}{\text{argmin}}\, L(\theta)$$

### Loss function: regression

- $y_i$ is scalar target variable for sample $i$, typically continuous values
- $z_{Oi}$ is estimate of $y_i$
- Use L2 loss:

$$L(\theta) = \sum_{i=1}^{N} (y_i - z_{Oi})^2$$

### Loss function: regression with vector output

- For vector $\mathbf{y}_i = (y_{i1}, \ldots, y_{iK})$, use vector L2 loss:

$$L(\theta) = \sum_{i=1}^{N} \sum_{i=1}^{K} (y_{iK} - z_{OiK})^2$$

### Loss function: binary classification (1)

- $y_i = 0, 1$ is target variable for sample $i$
- $z_{Oi}$ is scalar, called "logit score"
- Negative log likelihood loss:

$$L(\theta) = -\sum_{i=1}^{N} \ln P(y_i|\mathbf{x}_i, \theta), \quad P(y_i = 1|\mathbf{x}_i, \theta) = \frac{1}{1 + e^{-z_{Oi}}}$$

### Loss function: binary classification (2)

Loss (binary cross entropy):

$$L(\theta) = \sum_{i=1}^{N} -y_i z_{Oi} + \ln(1 + e^{y_i z_O i})$$

**Loss function: multi-class classification (1)**

- $y_i = 1, \ldots, K$ is target variable for sample $i$
- $\mathbf{z}_{Oi} = (z_{Oi1}, \ldots, z_{OiK})$ is vector with one entry per class, called "logit score"
- Likelihood given by softmax function, class with highest logit score has highest probability:

$$P(y_i = k|\mathbf{x}_i, \theta) = g_k(\mathbf{z}_{Oi}), \quad g_k(\mathbf{z}_{Oi}) = \frac{e^{z_{OiK}}}{\sum_\ell e^{z_{Oi\ell}}}$$

**Loss function: multi-class classification (2)**

Define "one-hot" vector - for a sample from class $k$, all entries in the vector are $0$ except for the $k$th entry which is $1$:

$$r_{ik} = \begin{cases} 1 & y_i = k \\ 0 & y_i \neq k \end{cases}$$

**Loss function: multi-class classification (3)**

Negative log likelihood

$$\ln P(y_i = k|\mathbf{x}_i, \theta) = \sum_{k=1}^{K} \ln P(y_i = k|\mathbf{x}_i, \theta)$$

**Loss function: multi-class classification (4)**

Loss (categorical cross entropy):

$$L(\theta) = \sum_{i=1}^{N} \left[ \ln \left( \sum_k e^{z_{Oik}} \right) - \sum_k r_{ik} z_{Oik} \right]$$

**Training: minimize loss function**

$$\hat{\theta} = \underset{\theta}{\text{argmin}}\, L(\theta), \quad L(\theta) = \frac{1}{N} \sum_{i=1}^{N} L_i(\theta, \mathbf{x}_i, y_i)$$

where $L_i(\theta, \mathbf{x}_i, y_i)$ is loss on sample $i$ for parameter $\theta$.

**Background: gradients of multi-variable functions**

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \partial L(\boldsymbol{\theta})/\partial \theta_1 \\ \vdots \\ \partial L(\boldsymbol{\theta})/\partial \theta_N \end{bmatrix}$$

### Gradients and optimization

Gradient has important properties for optimization:

- At a local minima (or maxima, or saddle point), $\nabla L(\theta) = 0$.
- At other points, $\nabla L(\theta)$ points towards direction of maximum (infinitesimal) *increase*.

### Gradient descent idea

To move towards minimum of a function, use first order approximation:

Start from some initial point, then iteratively

- compute gradient at current point, and
- add some fraction of the negative gradient to the current point

### Gradient descent illustration

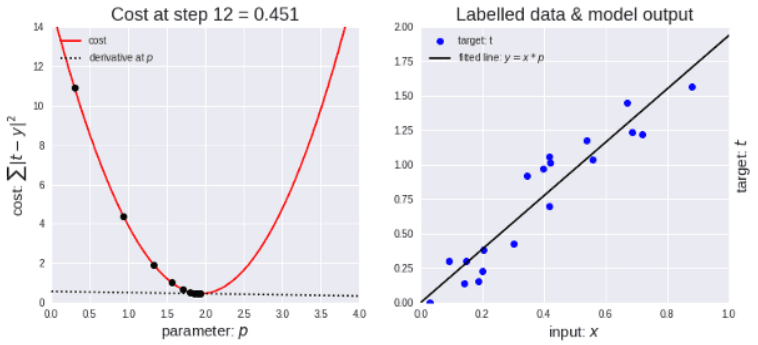



Figure 8: Link for animation. Image credit: Peter Roelants

### Standard ("batch") gradient descent

For each step $t$ along the error curve:

$$\theta^{t+1} = \theta^t - \alpha \nabla L(\theta^t) = \theta^t - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla L_i(\theta^t, \mathbf{x}_i, y_i)$$

Repeat until stopping criterion is met.

To update $\theta$, must compute $N$ loss functions and gradients - expensive when $N$ is large!

### Stochastic gradient descent

Idea: at each step, compute estimate of gradient using only one randomly selected sample, and move in the direction it indicates.

Many of the steps will be in the wrong direction, but progress towards minimum occurs *on average*, as long as the steps are small.

Bonus: helps escape local minima.

**Mini-batch (also "stochastic") gradient descent**

Idea: In each step, select a small subset of training data ("mini-batch"), and evaluate gradient on that mini-batch. Then move in the direction it indicates.

For each step $t$ along the error curve:

- Select random mini-batch $I_t \subset 1, \ldots, N$
- Compute gradient approximation: $g^t = \frac{1}{|I_t|} \sum_{i \in I_t} \nabla L(\mathbf{x}_i, y_i, \theta)$
- Update parameters: $\theta^{t+1} = \theta^t - \alpha^t g^t$

**Comparison: batch size**

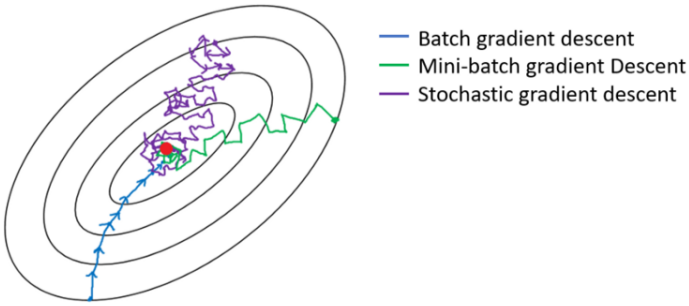



Figure 9: Effect of batch size on gradient descent.

**Why does mini-batch gradient help? (Intuition)**

- Standard error of mean over $m$ samples is $\frac{\sigma}{\sqrt{m}}$, where $\sigma$ is standard deviation.
- The benefit of more examples in reducing error is less than linear!
- Example: gradient based on 10,000 samples requires 100x more computation than one based on 100 samples, but reduces SE only 10x.
- Also: memory required scales with mini-batch size.
- Also: there is often redundancy in training set.

**Gradient descent terminology**

- Mini-batch size is $B$, training size is $N$
- A training *epoch* is the sequence of updates over which we see all non-overlapping mini-batches
- There are $\frac{N}{B}$ steps per training epoch
- Data shuffling: at the beginning of each epoch, randomly shuffle training samples. Then, select mini-batches in order from shuffled samples.

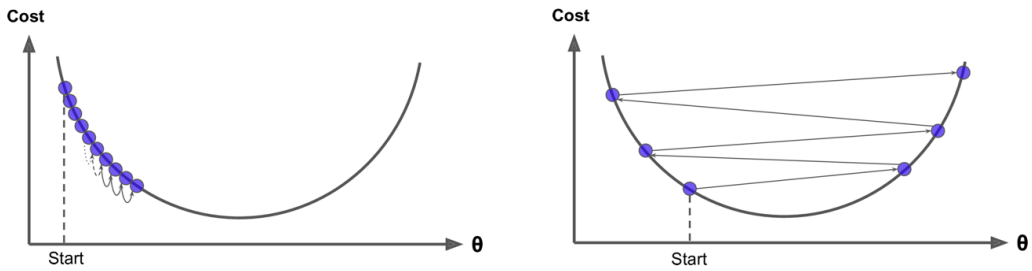
**Selecting the learning rate**

**Annealing the learning rate**

One approach: decay learning rate slowly over time, such as

- Exponential decay: $\alpha_t = \alpha_0 e^{-kt}$
- 1/t decay: $\alpha_t = \alpha_0/(1 + kt)$

(where $k$ is tuning parameter).




a) too small a) too big

Figure 10: Choice of learning rate $\alpha$ is critical

**Gradient descent in a ravine (1)**

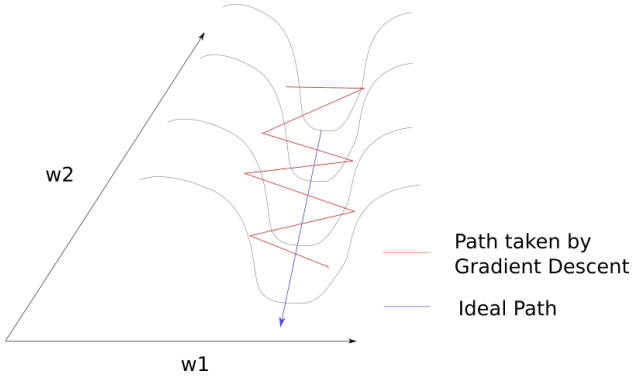



Figure 11: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of $w_1$.

**Gradient descent in a ravine (2)**

**Momentum (1)**

- Idea: Update includes a *velocity* vector $v$, that accumulates gradient of past steps.
- Each update is a linear combination of the gradient and the previous updates.
- (Go faster if gradient keeps pointing in the same direction!)

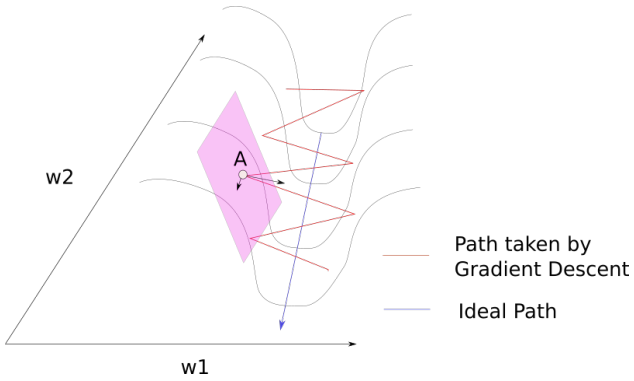



Figure 12: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of $w_1$.

**Momentum (2)**

Classical momentum: for some $0 \leq \gamma_t < 1$,

$$v_{t+1} = \gamma_t v_t - \alpha_t \nabla L\left(\theta_t\right)$$

so

$$\theta_{t+1} = \theta_t + v_{t+1} = \theta_t - \alpha_t \nabla L\left(\theta_t\right) + \gamma_t v_t$$

($\gamma_t$ is often around 0.9, or starts at 0.5 and anneals to 0.99 over many epochs.)

Note: $v_{t+1} = \theta_{t+1} - \theta_t$ is $\Delta\theta$.

**Momentum: illustrated**

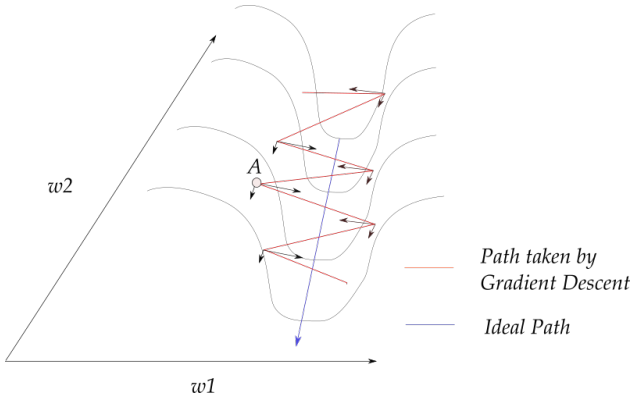



Figure 13: Momentum dampens oscillations by reinforcing the component along $w_2$ while canceling out the components along $w_1$.

**RMSProp**

Idea: Track *per-parameter* EWMA of *square* of gradient, and use to normalize parameter update step.

Weights with recent gradients or large magnitude have smaller learning rate, weights with small recent gradients have larger learning rates.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{\epsilon + E[g^2]_t}} \nabla L(\theta_{t,i})$$

where

$$E[g^2]_t = (1-\gamma)g^2 + \gamma E[g^2]_{t-1}, \quad g = \nabla J(\theta_{t,i})$$

**RMSProp: illustrated (Beale's function)**

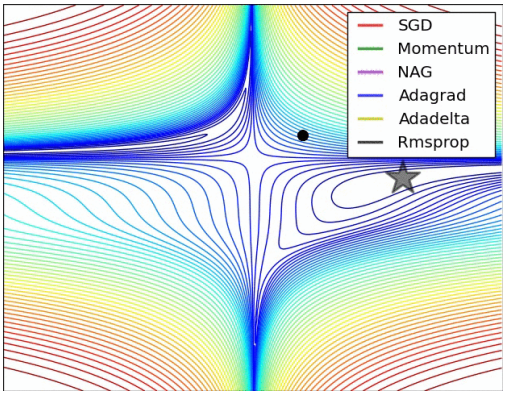



Figure 14: Animation credit: Alec Radford. Link for animation. Due to the large initial gradient, velocity based techniques shoot off and bounce around, while those that scale gradients/step sizes like RMSProp proceed more like accelerated SGD.

**RMSProp: illustrated (Long valley)**

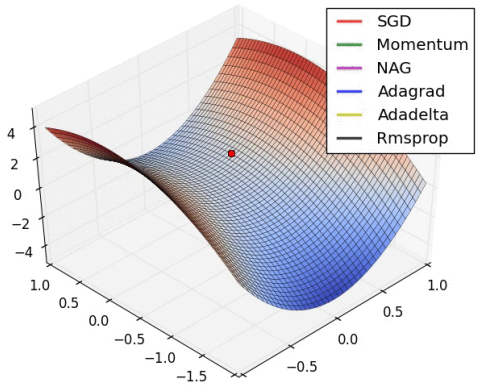



Figure 15: Animation credit: Alec Radford. Link for animation. SGD stalls and momentum has oscillations until it builds up velocity in optimization direction. Algorithms that scale step size quickly break symmetry and descend in optimization direction.

**Adam: Adaptive moments estimation (2014)**

Idea: Track the EWMA of *both* first and second moments of the gradient, $\{m_t, v_t\}$ at each time $t$.

If $L_t(\theta)$ is evaluation of loss function on a mini-batch of data at time $t$,

$$\{m_t, v_t\}, \mathbb{E}[m_t] \approx \mathbb{E}[\nabla L_t(\theta)], \mathbb{E}[v_t] \approx \mathbb{E}[(\nabla L_t(\theta))^2]$$

Scale $\alpha$ by $\frac{m_t}{\sqrt{v_t}}$ at each step.