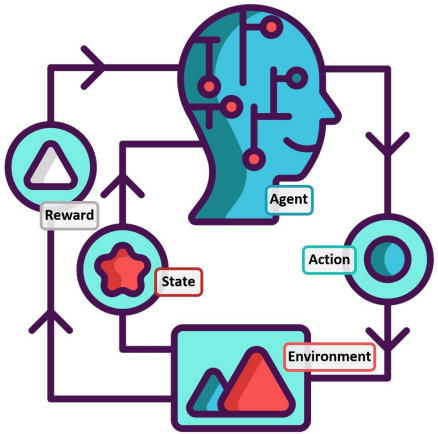
AMCO
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND CONTROL RESEARCH GROUP

Reinforcement Learning
2025/2026

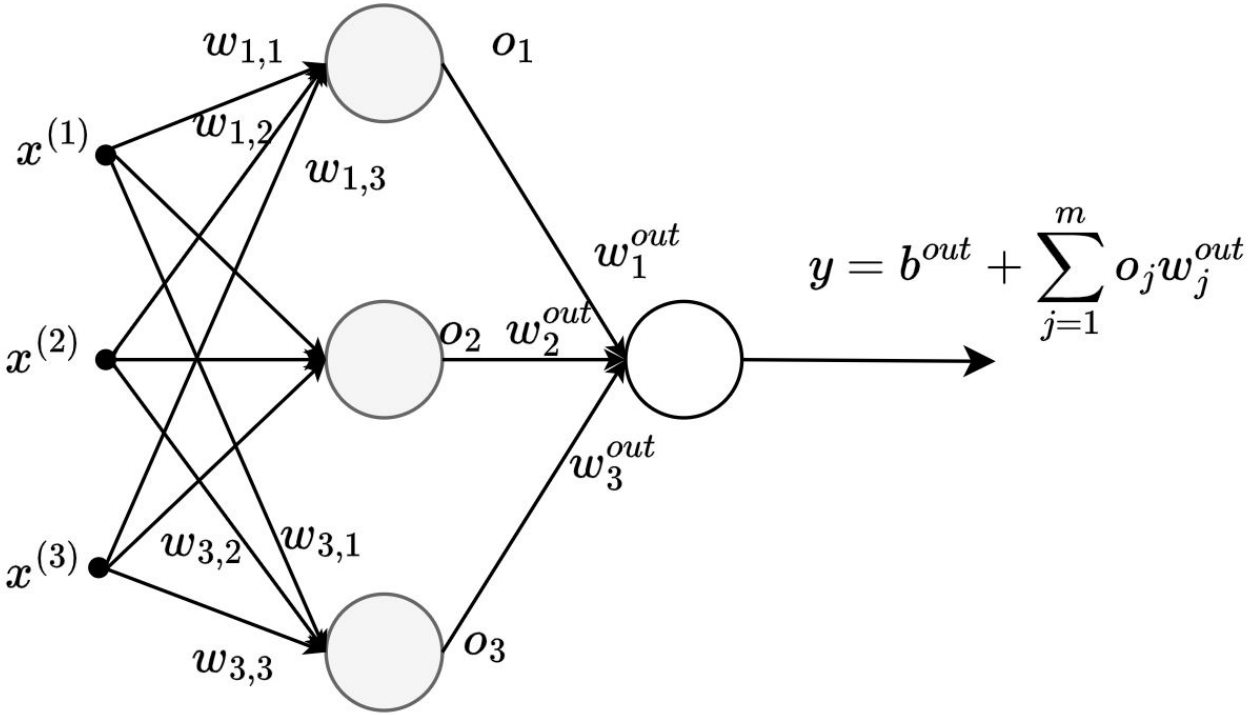
RL2
Reinforcement Learning Research Lab @ UniPD

Lecture #16
Neural Networks and Deep Learning



Riccardo De Monte
Gian Antonio Susto

Single-layer Feed Forward Neural Network



Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

0893-6080/89 $3.00 + .00

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KUR' HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

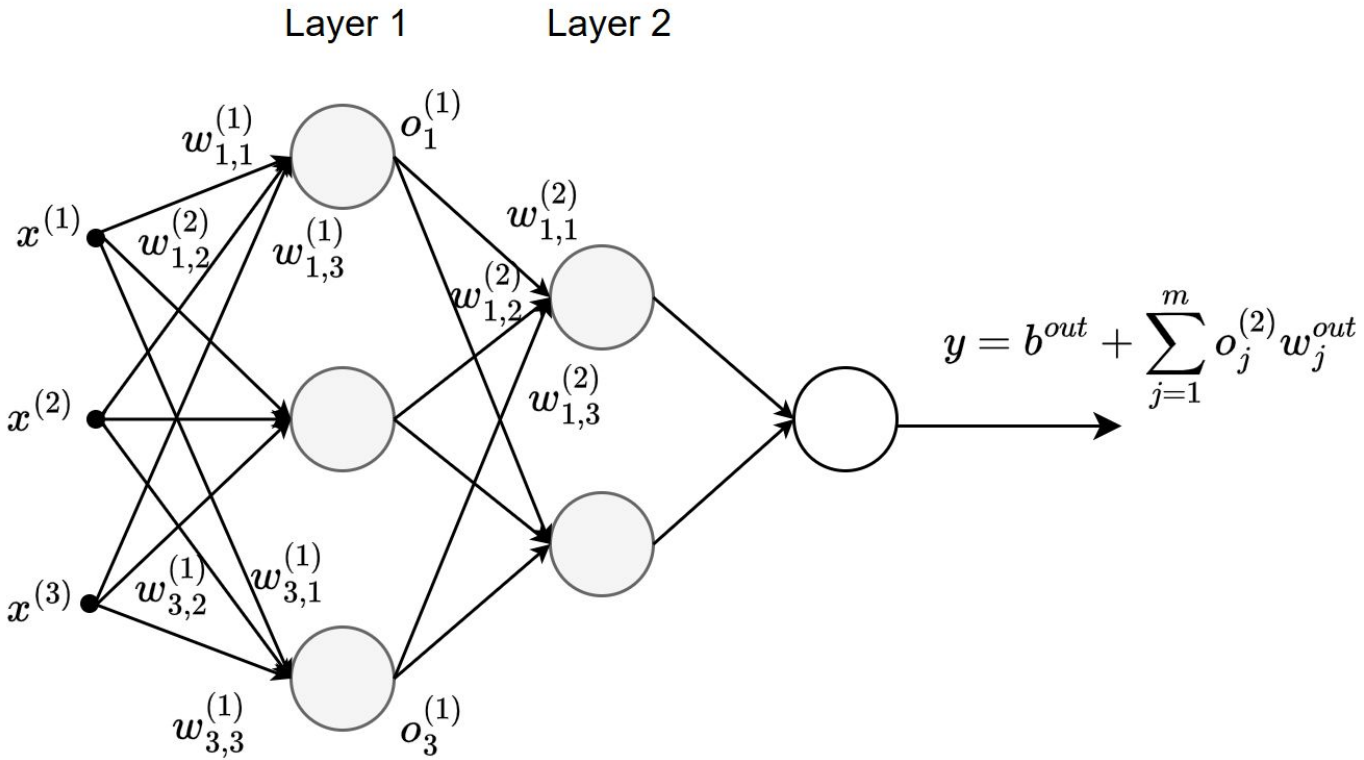
University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

Recall an example of a Deep NN



Matrix notation for multi-layer Neural Networks

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,m^{(l-1)}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,m^{(l-1)}}^{(l)} \\ \cdots & & & \\ w_{m^l,1}^{(l)} & w_{m^l,2}^{(l)} & \cdots & w_{m^l,m^{(l-1)}}^{(l)} \end{bmatrix} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}, \quad \boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{m^(l)}^{(l)} \end{bmatrix}$$

$$\boldsymbol{o}^{(1)} = g(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}), \qquad \boldsymbol{o}^{(l)} = g(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)})$$

Gradient Descent (GD), vector case

For Neural networks, we have

$$\mathcal{W} \in \mathbb{R}^D, \quad D \gg d$$

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

Here we consider the gradient (one derivative per parameter):

$$\nabla_{\mathcal{W}} \mathcal{L}(\mathcal{D}, \mathcal{W}) = \begin{bmatrix} \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,1}^{(1)}} \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,2}^{(1)}} \\ \vdots \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{w_{1,1}^{(L)}} \\ \vdots \\ \frac{\partial \mathcal{L}(\mathcal{D}, \mathcal{W})}{b^{(L)}} \end{bmatrix} \in \mathbb{R}^D$$

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \nabla_{\mathcal{W}} \mathcal{L}(\mathcal{D}, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}}$$

Gradient Descent is not practical

We cannot compute the gradient for the entire dataset

Stochastic Gradient Descent (SGD)

1. Recall that the overall loss function is an empirical average

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

Stochastic Gradient Descent (SGD)

1. Recall that the overall loss function is an empirical average

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, \mathcal{W})$$

2. A SGD update considers just one sample, randomly sampled:

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \nabla_{\mathcal{W}} l(x_i, y_i, \mathcal{W})\Big|_{\mathcal{W}=\mathcal{W}_{t-1}}, \qquad i \sim \{1, 2, \ldots, N\}$$

Does it make sense???

SGD is a noisy version of GD: we estimate the actual gradient with one sample. This estimate is unbiased:

$$\mathbb{E}_i \left[\nabla_{\mathcal{W}} l(x_i, y_i, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}} \right] = \sum_{i=1}^{N} \frac{1}{N} \nabla_{\mathcal{W}} l(x_i, y_i, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}} = \nabla_{\mathcal{W}} \mathcal{L}(\mathcal{D}, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}}$$

Is SGD just a cheap version of GD? (and unbiased)

My answer is: NO!

The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

My answer is: NO!

1. The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

2. There is a lot of theory about SGD and generalization, but here I just present my informal view.

My answer is: NO!

1. The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

2. There is a lot of theory about SGD and generalization, but here I just present my informal view.

3. We don’t apply SGD just to approximate GD. In fact, by linearity we can ideally accumulate gradient and then perform a GD.

My intuition

1. The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

2. I see SGD procedure, as a kind of statistical bootstrapping of samples form the true and unknown distribution:

My intuition

1. The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

2. I see SGD procedure, as a kind of statistical bootstrapping of samples form the true and unknown distribution:

    a. Each gradient used in SGD is an unbiased estimate of the gradient of the true risk.

My intuition

1. The “True” objective is not the Empirical Risk. Instead, we aim to minimize

$$\mathbb{E}_{(x,y)}\left[l(x, y, \mathcal{W})\right]$$

2. I see SGD procedure, as a kind of statistical bootstrapping of samples form the true and unknown distribution:

    a. Each gradient used in SGD is an unbiased estimate of the gradient of the true risk.

    b. Even if we have just a dataset, randomness allows to avoid to get stuck in minimum of the Empirical Risk.

Why SGD: recap

1. It is cheap.

2. Loss function are not convex w.r.t. network parameters: SGD might help.

3. Better generalization.

4. On-line updates: useful for Reinforcement Learning.

What we do in practice

Mini-batch SGD

Instead of one sample per update, we can use a batch of samples

$$B \subset \{1, 2, \ldots, N\}, \qquad \mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \frac{1}{|B|} \sum_{j \in B} \nabla_{\mathcal{W}} l(x_j, y_i, \mathcal{W}) \Big|_{\mathcal{W}=\mathcal{W}_{t-1}}$$

Random shuffling

Both for SGD and Mini-batch SGD, we don't actually sample, but instead we shuffle the indices of samples (e.g. from 1,2,3,4 -> 3,2,4,1) and we follow the new random order.

1. Most theory focuses on the random sampling approach.

2. Random shuffling is used in DL, but not always in Deep-RL (future lecture).

3. It guarantees that all samples are actually used.

Multiple passes: epochs

1. We introduced random shuffling to perform sampling without replacement.

Multiple passes: epochs

1. We introduced random shuffling to perform sampling without replacement.

2. To let multiple updates on each sample (like GD), once we have completed one update with all samples, we perform random shuffling again, and we repeat the update process.

Multiple passes: epochs

1. We introduced random shuffling to perform sampling without replacement.

2. To let multiple updates on each sample (like GD), once we have completed one update with all samples, we perform random shuffling again, and we repeat the update process.

3. **epochs = number of complete updates using the entire dataset**

Mini-batch SGD + random shuffling + epochs

```
# X and Y are lists
# model is the NN, model.W are the parameters
# loss_func is a loss function e.g. l2 loss
# grad_func is a function that computes the gradient

N = len(X)
epochs = 10
batch_size = 10
step_size = 0.03

for epoch in range(epochs):
    # Generate a random permutation of indices
    indices = np.random.permutation(N)

    num_batches = N // batch_size

    for batch_id in range(num_batches):
        grad = 0.0  # accumulate gradients over the batch

        # Process one mini-batch
        for i in range(batch_size):
            idx = indices[batch_id * batch_size + i]
            x, y = X[idx], Y[idx]

            output = model(x)
            loss = loss_func(y, output)
            grad += grad_func(loss, model)

        # Average the gradient over the batch
        grad /= batch_size

        # Parameter update
        model.W -= step_size * grad
```

How to compute the gradient?


BACKPROPAGATION
CHAIN RULE
imgflip.com

Chain Rule

$$f : \mathbb{R} \to \mathbb{R}, \quad g : \mathbb{R} \to \mathbb{R}$$

$$\frac{d}{dx} g(f(x)) = g'(f(x)) \cdot f'(x)$$

Toward backprop: part 1

$$f(x, w) = w \cdot x, \quad g : \mathbb{R} \to \mathbb{R}$$

$$\frac{d}{dw} g(f(x, w)) = \underbrace{\frac{d}{dw} f(x, w)}_{(1)} \cdot \underbrace{g'(f(x, w))}_{(2)} = x \cdot g'(x \cdot w)$$

Chain Rule: another example

$$f : \mathbb{R}^n \to \mathbb{R}, \quad g : \mathbb{R} \to \mathbb{R}$$

$$\nabla_x g(f(x)) = g'(f(x)) \cdot \nabla_x f(x)$$

Toward backprop: part 2

$$x, w \in \mathbb{R}^n, \quad f(x, w) = w^\top \cdot x, \quad g : \mathbb{R} \to \mathbb{R}$$

$$\nabla_w g(f(x, w)) = \underbrace{\nabla_w f(x, w)}_{(1)} \cdot \underbrace{g'(f(x, w))}_{(2)} = x \cdot g'(w^\top x)$$

Chain Rule: another example (might be confusing)

$$x \in \mathbb{R}^n, W \in \mathbb{R}^{m \times n} \quad f(x, W) = Wx, \quad g : \mathbb{R}^m \to \mathbb{R}$$

$$\nabla_W g(f(x, w)) = \underbrace{J_f(x, W)^\top}_{(1)} \cdot \underbrace{\nabla_f g(f(x, W))}_{(2)}$$

Take home message

1. Our main objective is to compute $\frac{\partial l(x, y, \mathcal{W})}{\partial w_{i,j}^{(l)}}$

Take home message

1. Our main objective is to compute $\dfrac{\partial l(x, y, \mathcal{W})}{\partial w_{i,j}^{(l)}}$

2. A NN is just a composition of many functions (many neurons, many layers, many activation functions).

$$\hat{y} = \mathrm{NN}(\boldsymbol{x}, \mathcal{W}) = \boldsymbol{f}^{(L)} \circ \boldsymbol{f}^{(L-1)} \circ \cdots \circ \boldsymbol{f}^{(1)}(\boldsymbol{x}) \quad \boldsymbol{f}^{(l)}(\boldsymbol{o}^{(l-1)}) = g\left(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

Take home message

1. Our main objective is to compute $\dfrac{\partial l(x, y, \mathcal{W})}{\partial w_{i,j}^{(l)}}$

2. A NN is just a composition of many functions (many neurons, many layers, many activation functions).

$$\hat{y} = \mathrm{NN}(\boldsymbol{x}, \mathcal{W}) = \boldsymbol{f}^{(L)} \circ \boldsymbol{f}^{(L-1)} \circ \cdots \circ \boldsymbol{f}^{(1)}(\boldsymbol{x}) \quad \boldsymbol{f}^{(l)}(\boldsymbol{o}^{(l-1)}) = g\left(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

3. We can apply the chain rule at any granularity level and to any composition of functions.

Take home message

1. Our main objective is to compute $\dfrac{\partial l(x, y, \mathcal{W})}{\partial w_{i,j}^{(l)}}$

2. A NN is just a composition of many functions (many neurons, many layers, many activation functions).

$$\hat{y} = \mathrm{NN}(\boldsymbol{x}, \mathcal{W}) = \boldsymbol{f}^{(L)} \circ \boldsymbol{f}^{(L-1)} \circ \cdots \circ \boldsymbol{f}^{(1)}(\boldsymbol{x}) \quad \boldsymbol{f}^{(l)}(\boldsymbol{o}^{(l-1)}) = g\left(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

3. We can apply the chain rule at any granularity level and to any composition of functions.

4. We need to go backward

The first derivative we need

Derivative of the loss w.r.t. the output of the NN:

$$\hat{y} = \mathrm{NN}(x, \mathcal{W}), \qquad l(x, y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

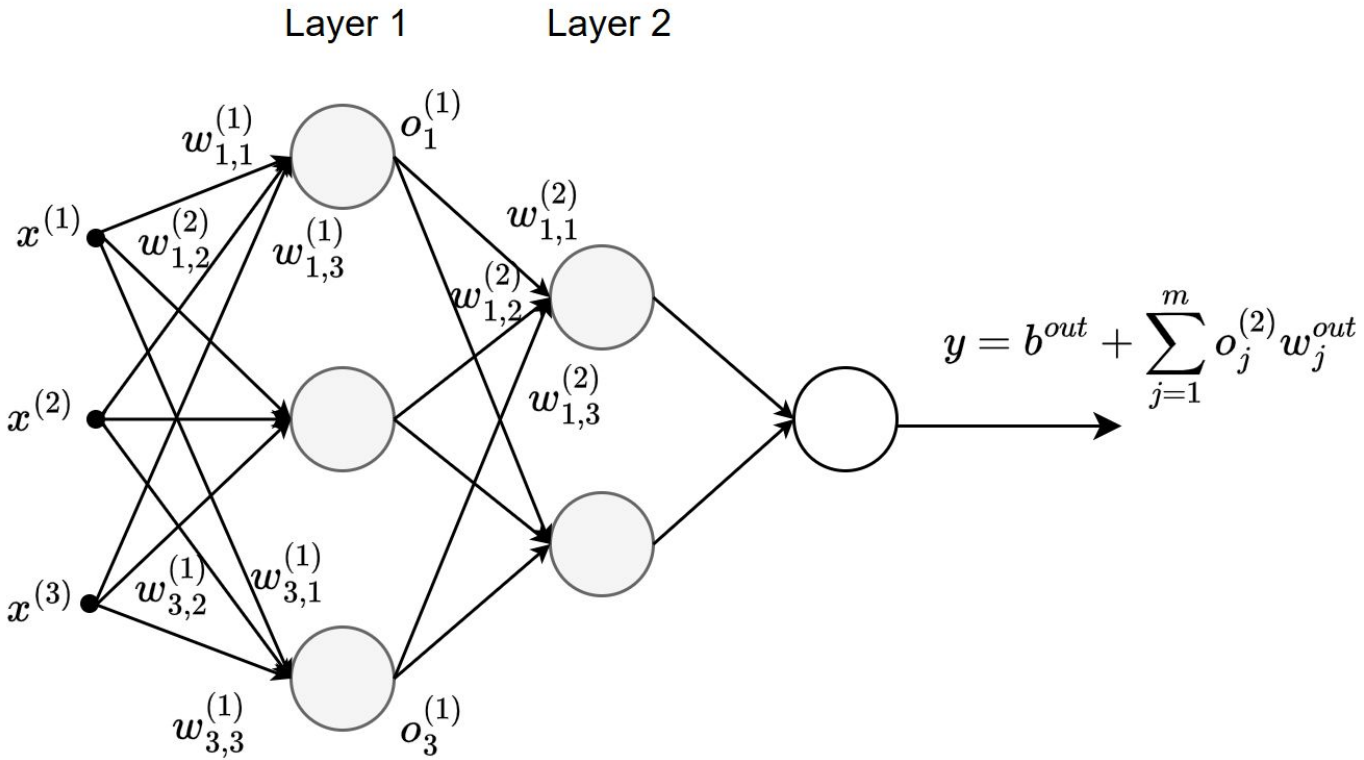
The first derivative we care of

Derivative of the loss w.r.t. the output of the NN:

$$\hat{y} = \mathrm{NN}(x, \mathcal{W}), \qquad l(x, y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial l}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \frac{1}{2}(y - \hat{y})^2 = (\hat{y} - y)$$

Recall an example of a Deep NN



Now we can go **backward**: for the output layer L

We can compute

$$\hat{y} = b^{(L)} + \sum_{i=1}^{m(L)} o_i^{(L-1)} w_i^{(L)}, \qquad \frac{\partial \hat{y}}{\partial w_j^{(L)}} = \frac{\partial}{\partial w_j^{(L)}} b^{(L)} + \sum_{i=1}^{m(L)} o_i^{(L-1)} \frac{\partial}{\partial w_j^{(L)}} w_i^{(L)} = o_j^{(L-1)}$$

Now we can go **backward**: for the output layer L

We can compute

$$\hat{y} = b^{(L)} + \sum_{i=1}^{m(L)} o_i^{(L-1)} w_i^{(L)}, \qquad \frac{\partial \hat{y}}{\partial w_j^{(L)}} = \frac{\partial}{\partial w_j^{(L)}} b^{(L)} + \sum_{i=1}^{m(L)} o_i^{(L-1)} \frac{\partial}{\partial w_j^{(L)}} w_i^{(L)} = o_j^{(L-1)}$$

For the chain rule

$$\frac{\partial l}{\partial w_j^{(L)}} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j^{(L)}} = o_j^{(L-1)} \cdot (\hat{y} - y)$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

$$\frac{\partial \hat{y}}{\partial o_j^{(L-1)}} = \frac{\partial}{\partial o_j^{(L-1)}} b^{(L)} + \sum_{i=1}^{m(L)} w_i^{(L)} \frac{\partial}{\partial o_j^{(L-1)}} o_i^{(L-1)} = w_j^{(L)}$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

$$\frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} = \frac{\partial o_j^{(L-1)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

$$\frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} = \frac{\partial o_j^{(L-1)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

For ReLU:

$$\frac{\partial o_j^{(L-1)}}{\partial a_j^{(L-1)}} = g'\left(a_j^{(L-1)}\right) = \begin{cases} a_j^{(L-1)} & a_j^{(L-1)} > 0 \\ 0 \end{cases}$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

$$\frac{\partial a_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} = \frac{\partial}{\partial w_{jk}^{(L-1)}} b_j^{(L-1)} + \sum_{p=1}^{m^{(L-1)}} o_p^{(L-2)} \frac{\partial}{\partial w_{jk}^{L-1}} w_{jp}^{(L-1)} = o_k^{(L-2)}$$

Now we can go **backward**: layer L–1

For the chain rule we have

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o_j^{(L-1)}} \cdot \frac{\partial o_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}$$

$$\frac{\partial l}{\partial w_{jk}^{(L-1)}} = (\hat{y} - y) \cdot w_j^{(L)} \cdot g'\left(a_j^{(L-1)}\right) \cdot o_k^{(L-2)}$$

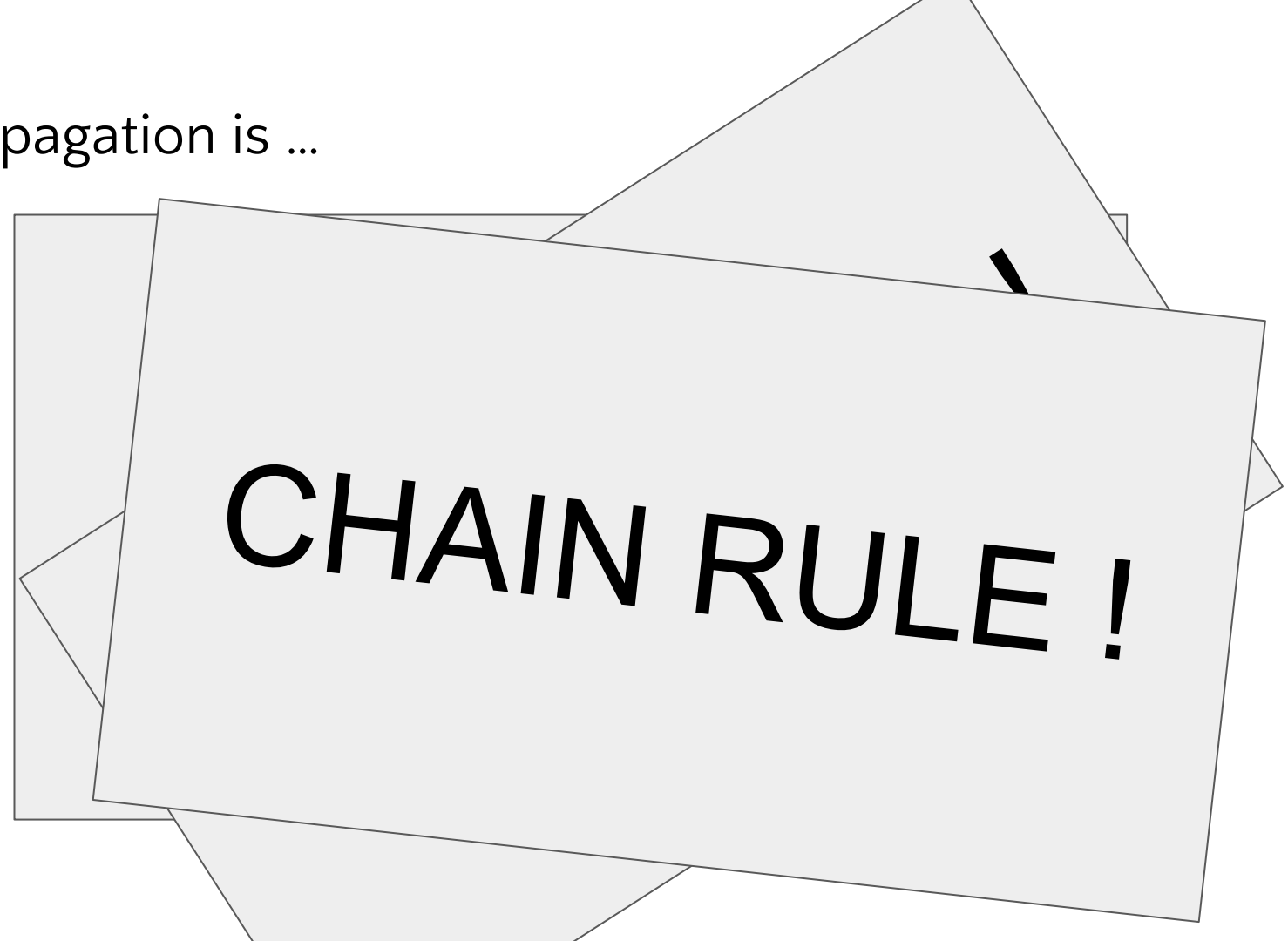
Backpropagation algorithm = apply the chain rule many times

Backpropagation is ...

Backpropagation is ...

Backpropagation is ...

Backpropagation is ...



PyTorch, Tensorflow, JAX implement this Automatic differentiation

Super useful video (and YT channel): use naive python to build an autodifferation engine

https://www.youtube.com/watch?v=VMj-3S1tku0


IT IS FRIDAY, ALL MY FRIENDS ARE AT PARTY
BUT, I'M HERE LEARNING BACK PROPAGATION
memegenerator.net

RECAP

1. Dataset D with sample (x,y)
2. We define a Deep NN.
   a. Initially the weights are randomly initialized.
   b. We should decide how many neurons, layers, which activation function
3. We define the loss function.
4. Mini-batch SGD updates (multiple epochs).
5. The gradient is computed using backpropagation.

A FFNN (MLP) is one type of NN. There are many others: CNNs, RNNs, Transformers, GNNs, SSMs, ...

Regularization

As for ML...

1. We have at least two datasets: a training one and a test one.

As for ML...

1. We have at least two datasets: a training one and a test one.

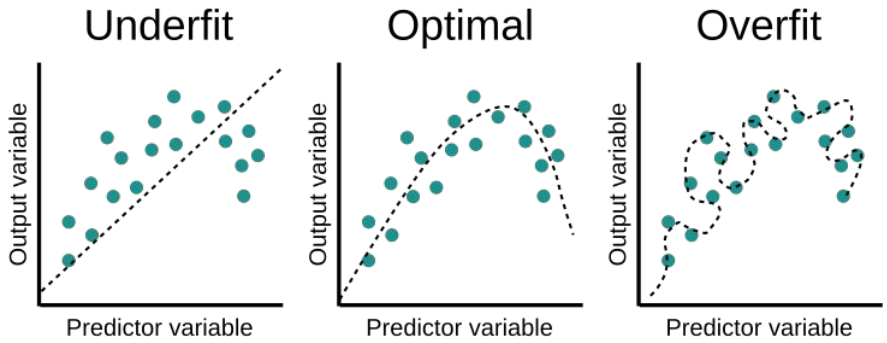
2. We use the test set to evaluate our model.

As for ML...

1. We have at least two datasets: a training one and a test one.

2. We use the test set to evaluate our model.

3. **Overfitting:**



Smooth NNs with L2 penalty

1. Overfitting => matching noise.

Smooth NNs with L2 penalty

1. Overfitting => matching noise.

2. We can avoid this by letting the NN smooth.

Smooth NNs with L2 penalty

1. Overfitting => matching noise.

2. We can avoid this by letting the NN smooth.

3. Smoothness = small variation in the input does not imply big changes of the output.

Smooth NNs with L2 penalty

1. Overfitting => matching noise.

2. We can avoid this by letting the NN smooth.

3. Smoothness = small variation in the input does not imply big changes of the output.

4. Derivative = local steepness

Smooth NNs with L2 penalty

1. For linear models:

$$y = w \cdot x + b, \qquad \frac{dy}{dx} = w$$

Smooth NNs with L2 penalty

1. For linear models:

$$y = w \cdot x + b, \qquad \frac{dy}{dx} = w$$

2. In deep learning we apply quite smooth non-linearities (ReLU).

Smooth NNs with L2 penalty

1. For linear models:

$$y = w \cdot x + b, \qquad \frac{dy}{dx} = w$$

2. In deep learning we apply quite smooth non-linearities (ReLU).

3. A NN is a composition of linear layer plus smooth activations.

Smooth NNs with L2 penalty

1. For linear models:

$$y = w \cdot x + b, \qquad \frac{dy}{dx} = w$$

2. In deep learning we apply quite smooth non-linearities (ReLU).

3. A NN is a composition of linear layer plus smooth activations.

4. We can just tune the parameters -> we should penalize the magnitude of weights.

Smooth NNs with L2 penalty

1. For linear models:

$$y = w \cdot x + b, \qquad \frac{dy}{dx} = w$$

2. In deep learning we apply quite smooth non-linearities (ReLU).

3. A NN is a composition of linear layer plus smooth activations.

4. We can just tune the parameters -> we should penalize the magnitude of weights.

5. As seen for backprop, derivatives of intermediate outputs w.r.t. inputs depend on weights.

Smooth NNs with L2 penalty

$$\mathcal{L}_{+reg}(\mathcal{D}, \mathcal{W}) = \mathcal{L}(\mathcal{D}, \mathcal{W}) + \lambda \underbrace{\sum_{l} \sum_{i} \sum_{j} \left| w_{i,j}^{(l)} \right|^2}_{||\mathcal{W}||_2^2}$$

Other stuffs

1. Early stopping.

2. Dropout: A Simple Way to Prevent Neural Networks from Overfitting.

3. Spectral Normalization (SN).

4. AdamW.

Additional stuffs about optimization

Stochastic Gradient Descent

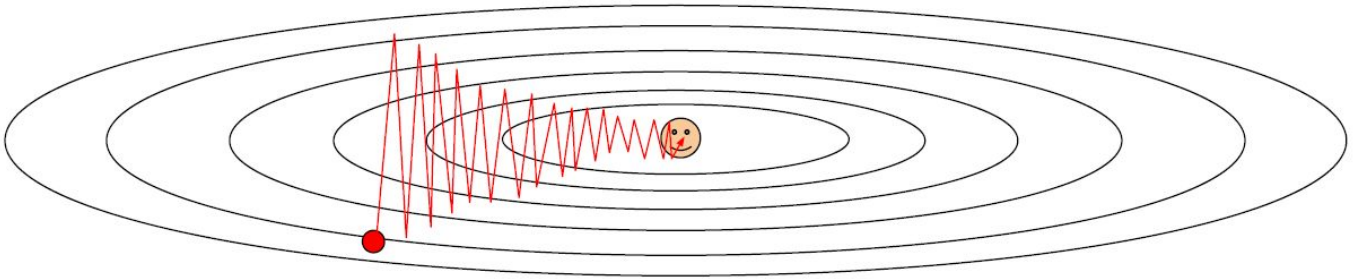
1. SGD

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \nabla_{\mathcal{W}} l(x_i, y_i, \mathcal{W})\Big|_{\mathcal{W}=\mathcal{W}_{t-1}}, \qquad i \sim \{1, 2, \ldots, N\}$$

2. Mini-batch SGD

$$B \subset \{1, 2, \ldots, N\}, \qquad \mathcal{W}_t = \mathcal{W}_{t-1} - \eta_t \frac{1}{|B|} \sum_{j \in B} \nabla_{\mathcal{W}} l(x_j, y_i, \mathcal{W})\Big|_{\mathcal{W}=\mathcal{W}_{t-1}}$$

Problem of SGD



F.-F. Li et A. *Convolutional Neural Networks for Visual Recognition* http://cs231n.stanford.edu/

SGDM: SGD with Momentum

1. We denote the gradient for one SGD/Mini-batch SGD step as:

$$g_t = \nabla_{\mathcal{W}} L(\mathcal{W}_{t-1}), \quad L = l_i \ \text{ or } \ L = \frac{1}{|B|} \sum_i l_i$$

SGDM: SGD with Momentum

1. We denote the gradient for one SGD/Mini-batch SGD step as:

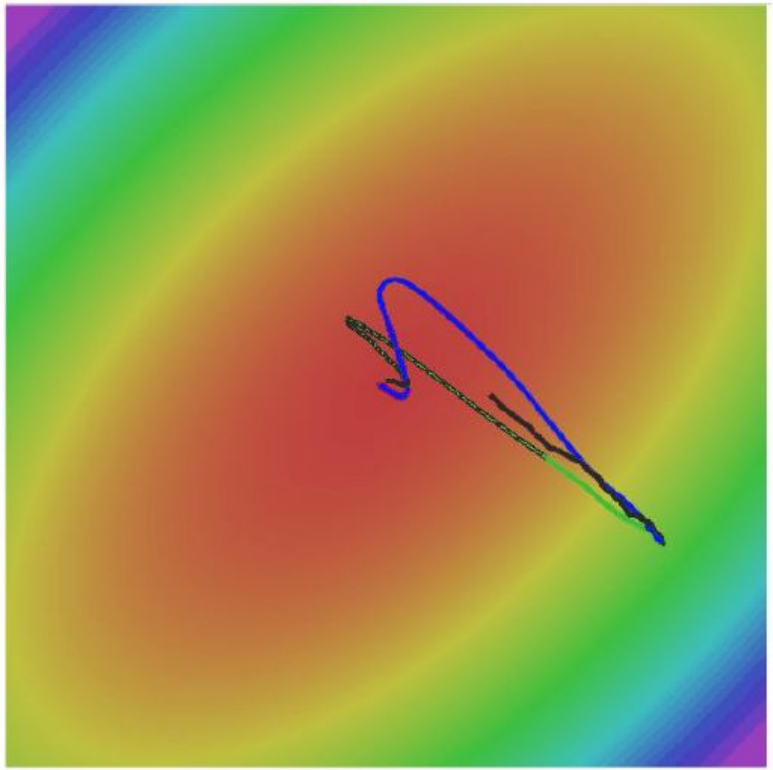
$$g_t = \nabla_{\mathcal{W}} L(\mathcal{W}_{t-1}), \quad L = l_i \ \text{ or } \ L = \frac{1}{|B|}\sum_i l_i$$

2. Heavy ball approach: for each parameter we keep a (exponential) moving average of the gradient.

$$m_t = \beta m_{t-1} + g_t \qquad \mathcal{W}_t = \mathcal{W}_{t-1} - \eta m_t$$

3. Beta is typically 0.9.

SGDM: SGD with Momentum



SGD

SGD+Momentum

Nesterov

F.-F. Li et A. *Convolutional Neural Networks for Visual Recognition* http://cs231n.stanford.edu/

SGDM: SGD with Momentum

1. Momentum accelerates convergence.

SGDM: SGD with Momentum

1. Momentum accelerates convergence.

2. Nesterov-momentum is an alternative.

SGDM: SGD with Momentum

1. Momentum accelerates convergence.

2. Nesterov-momentum is an alternative.

3. Momentum reduces variance of SGD: it acts as a low-pass filter.

SGDM: SGD with Momentum

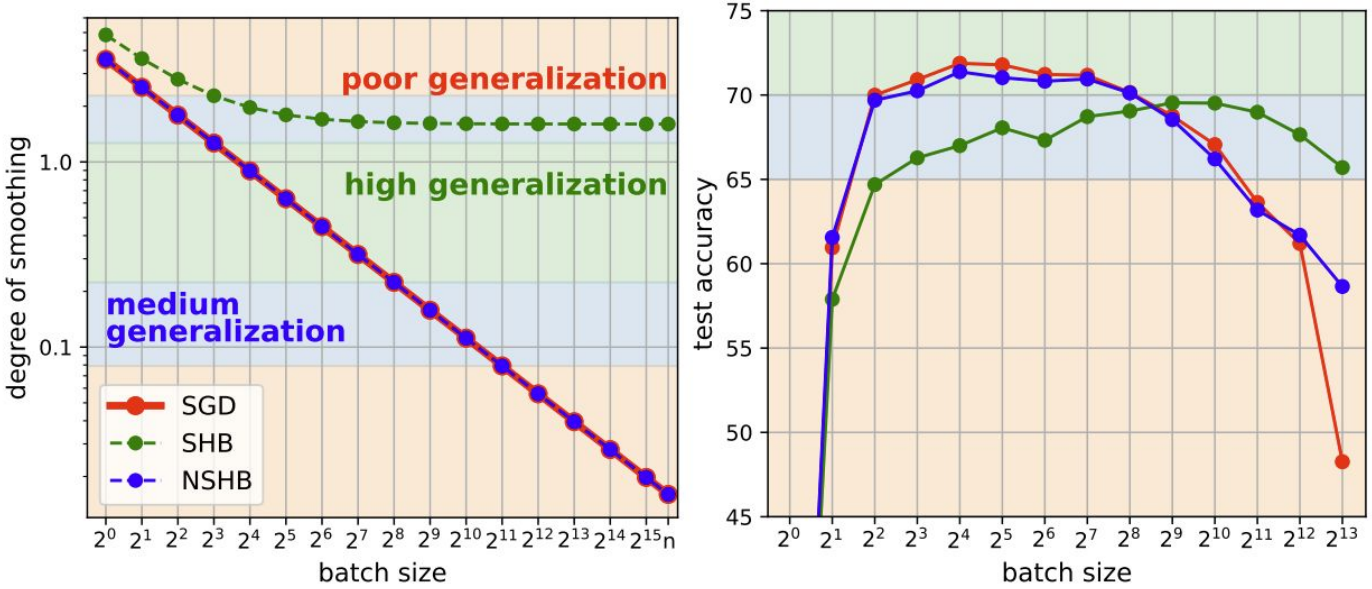
1. Momentum accelerates convergence.

2. Nesterov-momentum is an alternative.

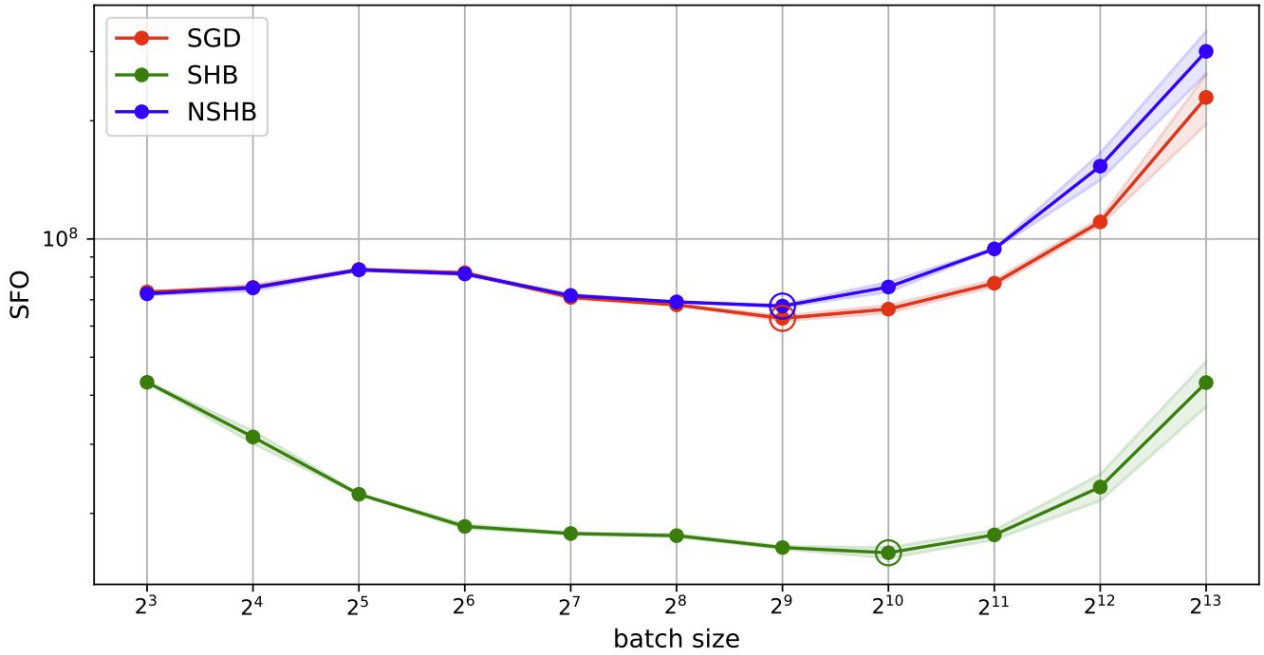
3. Momentum reduces variance of SGD: it acts as a low-pass filter.

4. By adding inertia, allows to escape from saddle points (useful for non-convex problems).

SGDM: SGD with Momentum



SGDM: SGD with Momentum



RMSProp

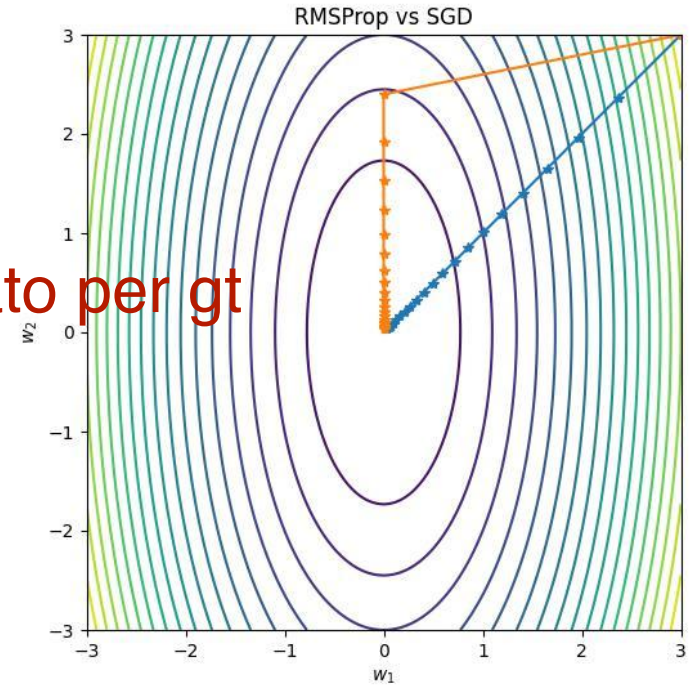
1. Root Mean Squared Propagation prevents too large steps for some parameters.

$$v_t = \alpha v_{t-1} + (1-\alpha) g_t^2$$

$$W_t = W_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon}$$

moltiplicato per gt

$$\alpha \in (0,1), \quad \epsilon = 10^{-7}$$



RMSProp

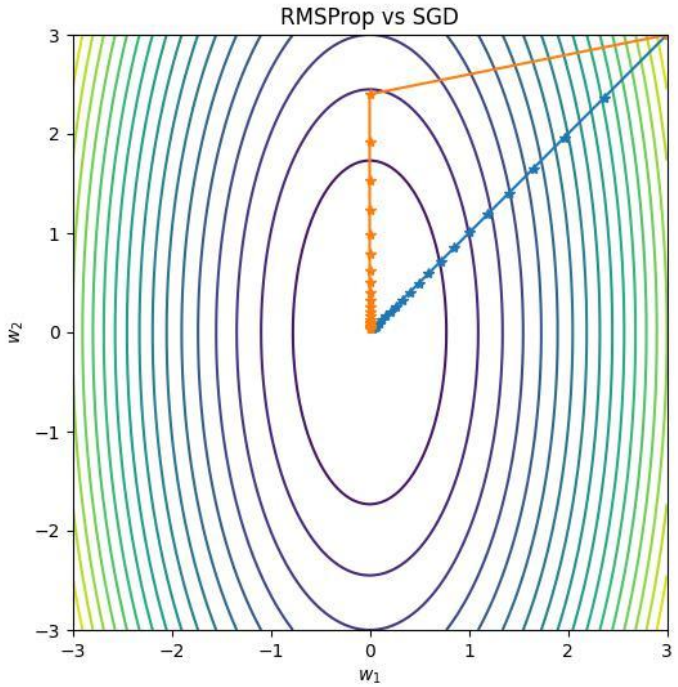
1. Root Mean Squared Propagation prevents too large steps for some parameters.

$$v_t = \alpha v_{t-1} + (1-\alpha) g_t^2$$

$$W_t = W_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon}$$

$$\alpha \in (0,1), \quad \epsilon = 10^{-7}$$

2. Each derivative is normalized: parameters with historically large gradients receive smaller updates, and those with small gradients receive larger updates.



Adam: RMSProp + Momentum + Bias Correction

$$m_t = \beta_1 m_{t_1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t_1} + (1 - \beta_2) g_t^2$$

$$\mathcal{W}_t = \mathcal{W}_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

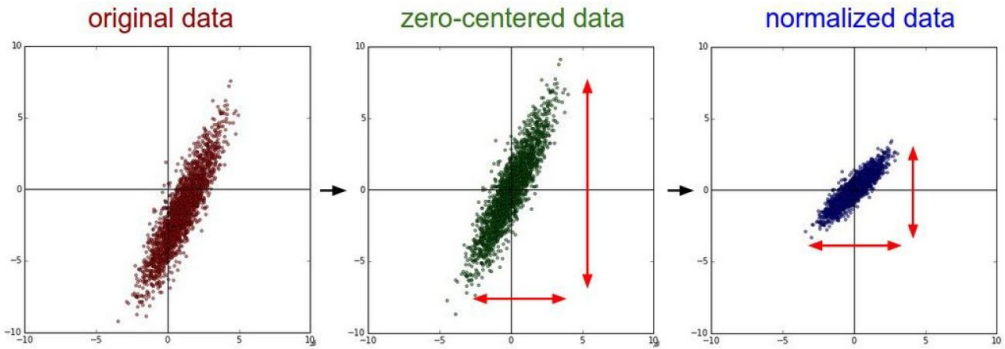
$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

beta_1 = 0.9, beta_2 = 0.999

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

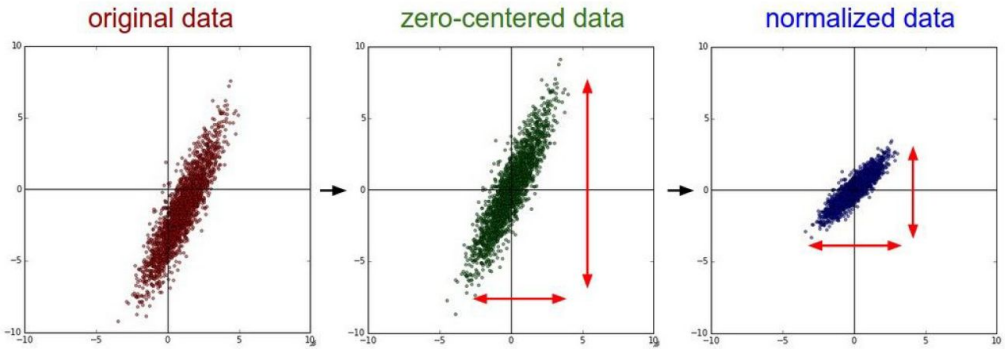
Normalization

Input normalization



1. As for ML, we would like input features in the same scale.

Input normalization



1. As for ML, we would like input features in the same scale.
2. Big input scales -> big gradients -> unstable training.
3. Small input scales -> vanishing gradients -> slow training.

Deep NNs have two or more layers

Weights initialization

1. Even if the input is normalized, we can still observe big/small gradients: gradients depend on both weights and intermediate outputs.

2. We should initialize parameters properly:
   a. Random initialization: each neuron has a different “purpose” (it should learn a different feature).
   b. Intermediate outputs should preserve variance of inputs. Same for gradient.
   c. To satisfy b, initialization usually depends on input/output size.

3. Example (used for ReLU activation function): He/Kaiming initialization

$$w_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{m^{(l-1)}}\right)$$

Internal covariate shift: the distribution of a layer’s inputs changes as the network learns

LayerNorm

Consider z any m-dimensional output of any layer. LayerNorm performs:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z_i, \quad \sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(z_i - \mu)^2}, \qquad \bar{z_i} = \frac{z_i - \mu}{\sigma} \cdot g_i + b_i$$

g_i and b_i are learnable.

LayerNorm

1. All transformer-based architectures use LayerNorm (or similar).

2. BatchNorm is more popular in vision, but LayerNorm works well too.

What about NNs for RL?

Value function

1. Recall the value function:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{i=t}^{+\infty} \gamma^{i-t} r_{i+1} \middle| s_t = s \right], \quad \Longrightarrow \quad v_\pi(\cdot) : \mathcal{S} \subseteq \mathbb{R}^d \to \mathbb{R}$$

Value function

1. Recall the value function:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{i=t}^{+\infty} \gamma^{i-t} r_{i+1} \middle| s_t = s \right], \quad \Longrightarrow \quad v_\pi(\cdot) : \mathcal{S} \subseteq \mathbb{R}^d \to \mathbb{R}$$

2. The return is a noisy measurement of the value function and its expected value is the value function:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \middle| s_t = s \right]$$

NN and Value function

1. We use NN to approximate the value function (NNs are universal approximator):

$$\mathrm{NN}(\boldsymbol{s}; \mathcal{W}) = \hat{v}(\boldsymbol{s}; \boldsymbol{w}) \approx v_\pi(\boldsymbol{s})$$

NN and Value function

1. We use NN to approximate the value function (NNs are universal approximator):

$$\mathrm{NN}(\boldsymbol{s};\mathcal{W}) = \hat{v}(\boldsymbol{s};\boldsymbol{w}) \approx v_\pi(\boldsymbol{s})$$

2. In this case samples (x,y) are (s_t, G_t). A possible loss function might be:

$$l(\boldsymbol{s}_t, G_t, \mathcal{W}) = \frac{1}{2}(G_t - \mathrm{NN}(\boldsymbol{s}_t;\mathcal{W}))^2$$

For control we do the same for Q(s,a)

Is it so simple?

In RL we have no fixed dataset

1. Even if we aim to just to prediction (given a fixed policy, learn v(s)), we do not have any fixed dataset.

In RL we have no fixed dataset

1. Even if we aim to just to prediction (given a fixed policy, learn v(s)), we do not have any fixed dataset.

2. When we perform control, Q(s,a) changes accordingly to the new policy: the NN has to approximate a new Q -> **CONTINUAL LEARNING**

Other stuffs

1. G_t is highly noisy -> highly noisy target -> noisy updates (using gradient)

Other stuffs

1. G_t is highly noisy -> highly noisy target -> noisy updates (using gradient)

2. We can use bootstrapping, but recall that we use the same NN to compute the target:

$$l(\boldsymbol{s}_t, r_{t+1}, \boldsymbol{s}_{t+1}, \mathcal{W}) = \frac{1}{2}(r_{t+1} + \gamma \mathrm{NN}(\boldsymbol{s}_{t+1}; \mathcal{W}) - \mathrm{NN}(\boldsymbol{s}_t; \mathcal{W}))^2$$