Insight
Centre for Data Analytics

DEEP
LEARNING
WORKSHOP

Dublin City University
21-22 May 2018

DCU

Day 1

Training Deep Networks with Backprop

Kevin McGuinness
kevin.mcguinness@dcu.ie

Assistant Professor
School of Electronic Engineering
Dublin City University

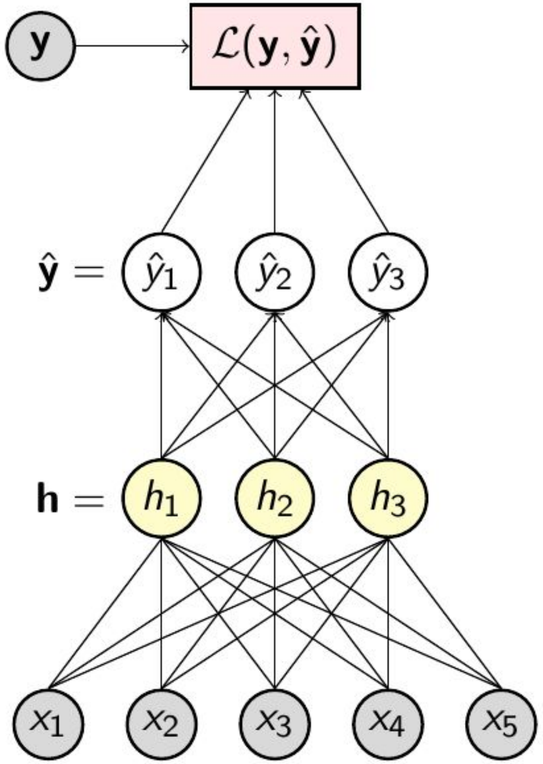
Recall: multi-layer perceptron

Input layer: $\{x_1, \ldots, x_D\}$

Hidden layer: $\{h_1, \ldots, h_M\}$

Output layer: $\{\hat{y}_1, \ldots, \hat{y}_K\}$

Loss: $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$



Recall: multi-layer perceptron

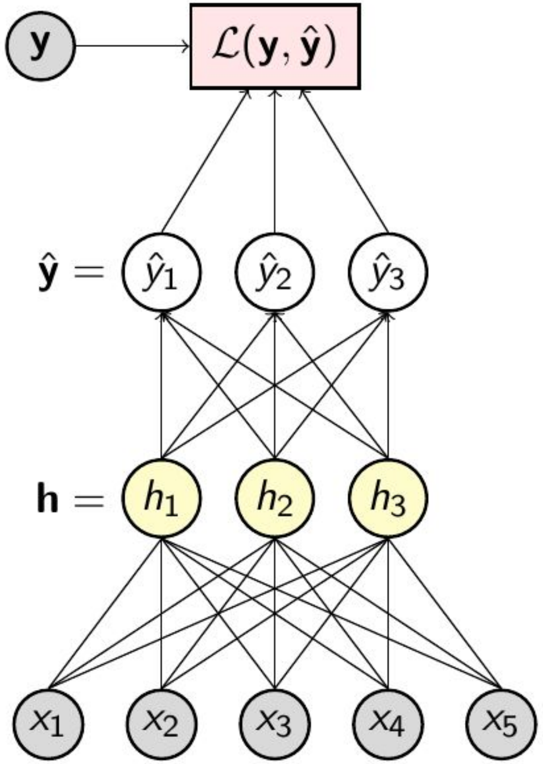
Input to hidden:

$$\mathbf{h} = g_1(W_1\mathbf{x} + \mathbf{b}_1)$$

Hidden to output:

$$\hat{\mathbf{y}} = g_2(W_2\mathbf{h} + \mathbf{b}_2)$$

g_1 and g_2 are **activation functions**. E.g. sigmoid.

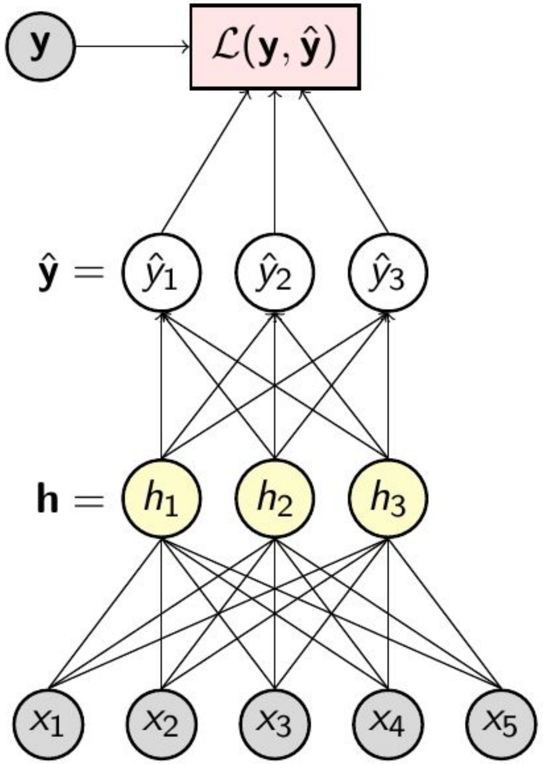


Recall: multi-layer perceptron

Decision function:

$$\begin{aligned}\hat{\mathbf{y}} &= f(\mathbf{x}) \\ &= g_2(W_2\mathbf{h} + \mathbf{b}_2) \\ &= g_2(W_2 g_1(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)\end{aligned}$$

If g_1 is a non-linear activation function (like a sigmoid), then $f(\mathbf{x})$ is a non-linear function of $\mathbf{x}$!



Fitting deep networks to data

We need an algorithm to find **good weight configurations**.

This is an unconstrained continuous **optimization problem**.

We can use standard iterative optimization methods like **gradient descent**.

To use gradient descent, we need a way to find the **gradient of the loss with respect to the parameters** (weights and biases) of the network.

Error **backpropagation** is an efficient algorithm for finding these gradients.

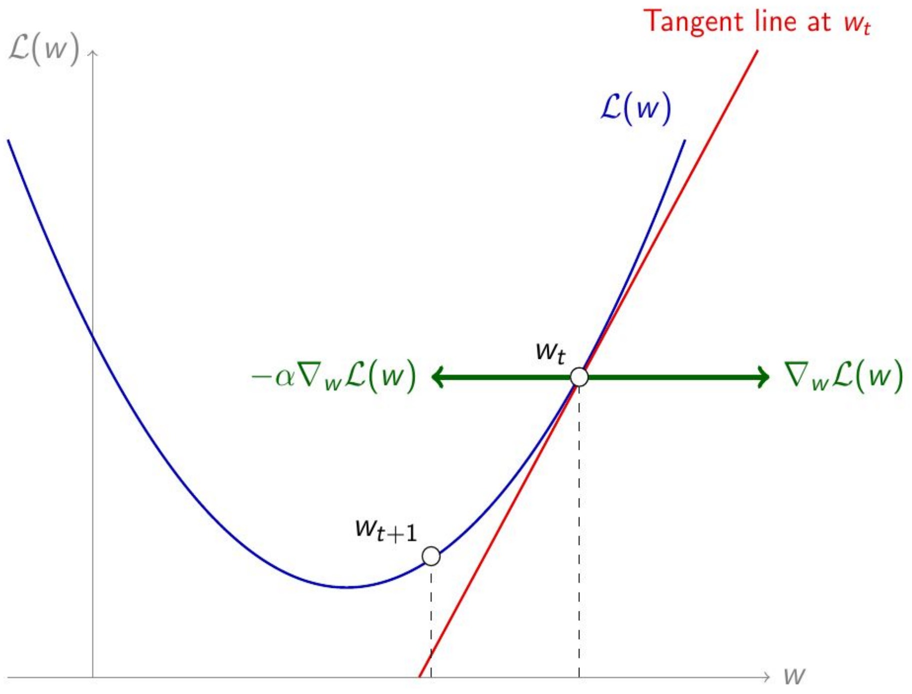
Basically an application of the multivariate **chain rule** and **dynamic programming**.

In practice, computing the full gradient is expensive. Backpropagation is typically used with **stochastic gradient descent**.

Gradient descent

If we had a way to compute the gradient of the loss with respect to the parameters, we could use gradient descent to optimize

$$W_1 \leftarrow W_1 - \alpha \nabla_{W_1} \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$$



Stochastic gradient descent

Computing the gradient for the full dataset at each step is slow

- Especially if the dataset is large!

For most losses we care about, the total loss can be expressed as a sum (or average) of losses on the individual examples

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

The gradient is the average of the gradients on individual examples

$$\nabla\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \nabla L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

Stochastic gradient descent

SGD: estimate the gradient using a subset of the examples

- Pick a **single random training example**
- **Estimate** a (noisy) loss on this single training example (the *stochastic* gradient)
- Compute gradient wrt. this loss
- Take a step of gradient descent using the estimated loss

Stochastic gradient descent

Advantages

- Very fast (only need to compute gradient on single example)
- Memory efficient (does not need the full dataset to compute gradient)
- Online (don’t need full dataset at each step)

Disadvantages

- Gradient is very noisy, may not always point in correct direction
- Convergence can be slower

Improvement

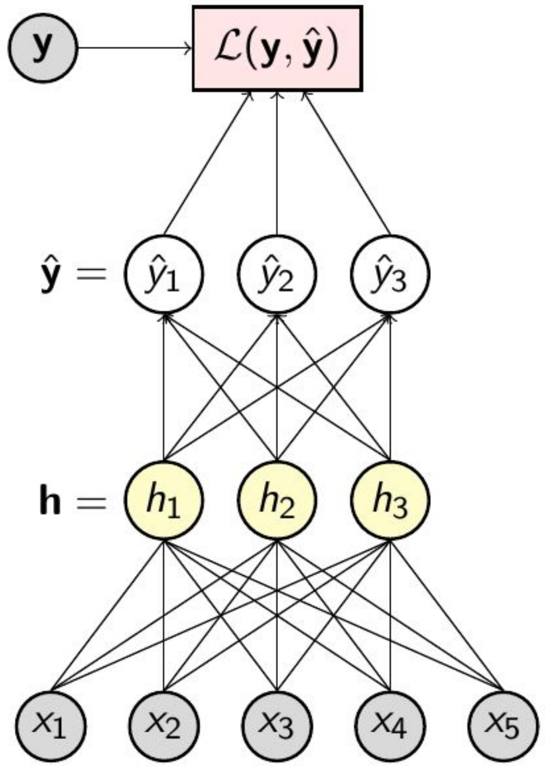
- Estimate gradient on small batch of training examples (say 50)
- Known as **mini-batch stochastic gradient descent**

Finding the gradient with backprop

Combination of the chain rule and dynamic programming

Chain rule: allows us to find gradient of the loss with respect to any input, activation, or parameter

Dynamic programming: reuse computations from previous steps. You don't need to evaluate the full chain for every parameter.



The chain rule

Easily differentiate compositions of functions.

$$y = f(z)$$
$$z = g(x)$$

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$$

The chain rule

$$\hat{y} = g_4(g_3(g_2(g_1(x_1))))$$

$$\begin{tikzcd} x_1 \arrow[r] & g_1 \arrow[r, "x_2"] & g_2 \arrow[r, "x_3"] & g_3 \arrow[r, "x_4"] & g_4 \arrow[r, "x_5"] & \hat{y} \end{tikzcd}$$

Decompose into steps (**forward propagation**):

$$\begin{aligned} x_2 &= g_1(x_1) \\ x_3 &= g_2(x_2) \\ x_4 &= g_3(x_3) \\ \hat{y} = x_5 &= g_4(x_4) \end{aligned}$$

The chain rule

$$\hat{y} = g_4(g_3(g_2(g_1(x_1))))$$

$$\begin{tikzcd}
x_1 \arrow[r] & g_1 \arrow[r, "x_2"] & g_2 \arrow[r, "x_3"] \arrow[l, bend left, "\delta_2"] & g_3 \arrow[r, "x_4"] \arrow[l, bend left, "\delta_3"] & g_4 \arrow[r, "x_5"] \arrow[l, bend left, "\delta_4"] & \hat{y} \arrow[l, bend left, "\delta_5"]
\end{tikzcd}$$

Want to find $\frac{\partial \hat{y}}{\partial x_1}$. Chain rule:

$$\frac{\partial \hat{y}}{\partial x_1} = \frac{\partial \hat{y}}{\partial x_4} \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x_1}$$

The chain rule

$$\begin{tikzcd}
x_1 \arrow[r] & g_1 \arrow[r, "x_2"] & g_2 \arrow[r, "x_3"] \arrow[l, bend left, "\delta_2"] & g_3 \arrow[r, "x_4"] \arrow[l, bend left, "\delta_3"] & g_4 \arrow[r, "x_5"] \arrow[l, bend left, "\delta_4"] & \hat{y} \arrow[l, bend left, "\delta_5"]
\end{tikzcd}$$

Decompose into steps again. Let $\delta_k = \frac{\partial \hat{y}}{\partial x_k}$. **Backpropagation**:

$$\begin{aligned}
\delta_5 &= \frac{\partial \hat{y}}{\partial x_5} = 1 \\
\delta_4 &= \frac{\partial \hat{y}}{\partial x_4} = \frac{\partial \hat{y}}{\partial x_5}\frac{\partial x_5}{\partial x_4} = \delta_5 g_4'(x_4) \\
\delta_3 &= \frac{\partial \hat{y}}{\partial x_3} = \frac{\partial \hat{y}}{\partial x_4}\frac{\partial x_4}{\partial x_3} = \delta_4 g_3'(x_3) \\
\delta_2 &= \frac{\partial \hat{y}}{\partial x_2} = \frac{\partial \hat{y}}{\partial x_3}\frac{\partial x_3}{\partial x_2} = \delta_3 g_2'(x_2) \\
\delta_1 &= \frac{\partial \hat{y}}{\partial x_1} = \frac{\partial \hat{y}}{\partial x_2}\frac{\partial x_2}{\partial x_1} = \delta_2 g_1'(x_1)
\end{aligned}$$

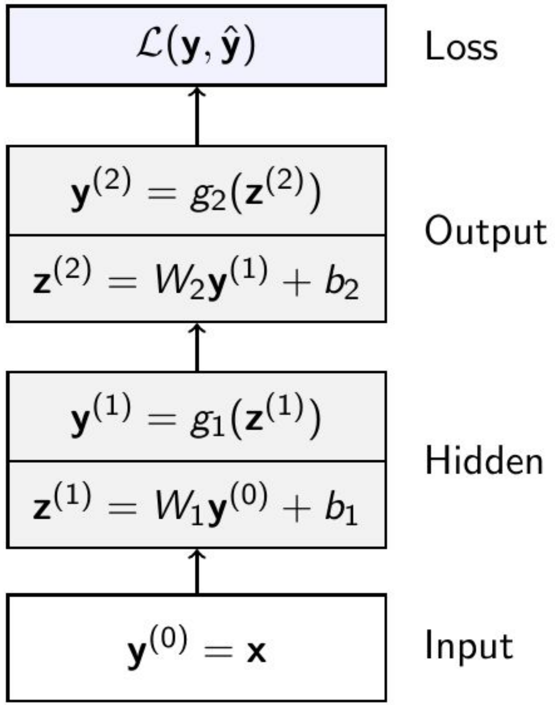
Dynamic programming

Need to calculate gradients wrt. parameters:

$$\frac{\partial \mathcal{L}}{\partial W_2} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\partial \mathbf{z}^{(2)}} \right) \frac{\partial \mathbf{z}^{(2)}}{\partial W_2}$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(2)}} \frac{\partial \mathbf{y}^{(2)}}{\partial \mathbf{z}^{(2)}} \right) \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{y}^{(1)}} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial W_1}$$

- Can re-use the bit in parenthesis when computing gradient wrt. W_1.



$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$
Loss
$\mathbf{y}^{(2)} = g_2(\mathbf{z}^{(2)})$
$\mathbf{z}^{(2)} = W_2\mathbf{y}^{(1)} + b_2$
Output
$\mathbf{y}^{(1)} = g_1(\mathbf{z}^{(1)})$
$\mathbf{z}^{(1)} = W_1\mathbf{y}^{(0)} + b_1$
Hidden
$\mathbf{y}^{(0)} = \mathbf{x}$
Input

Modular backprop

You could use the chain rule on all the individual neurons to compute the gradients with respect to the parameters and backpropagate the error signal.

Much more useful to use the **layer abstraction**

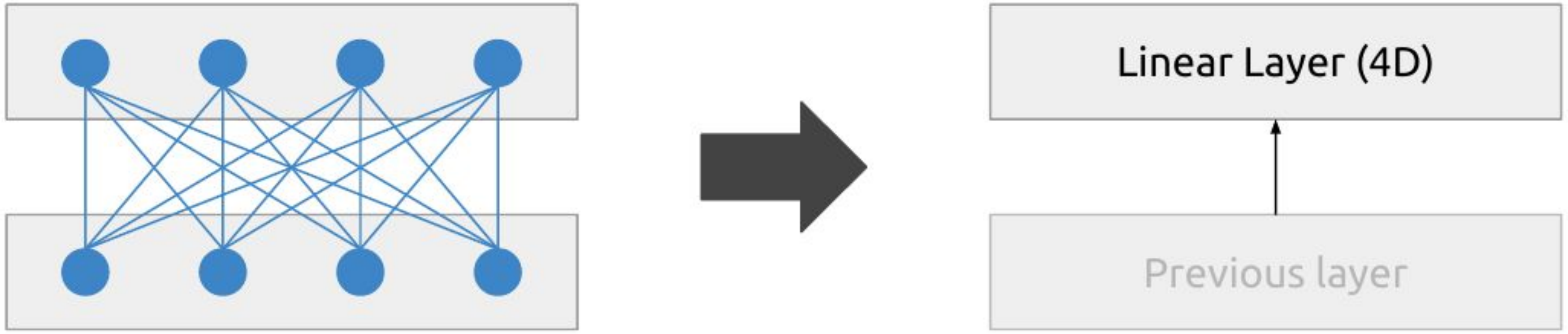
Then define the backpropation algorithm in terms of three operations that layers need to be able to do.

This is called **modular backpropagation**

The layer abstraction

E.g. a linear layer. Each output is a linear combination of its inputs (plus a bias). The linear layer corresponds to a matrix multiplication by the weights:

$$\mathbf{x}_{t+1} = W\mathbf{x}_t + \mathbf{b}$$

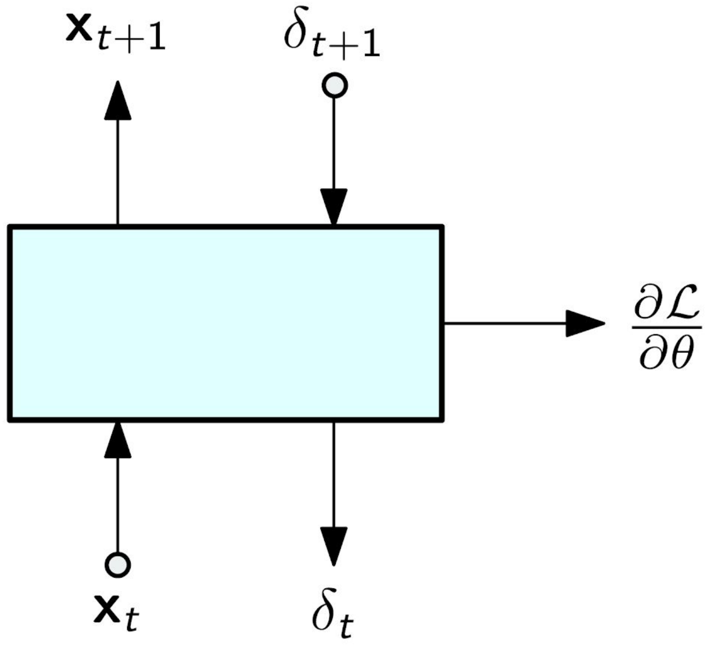


The layer abstraction

Need to update our layer abstraction to include functionality for back propagation.

Each layer to implement three functions:

1. **forward**: compute $\mathbf{x}_{t+1}$ from $\mathbf{x}_t$
2. **backward**: compute δ_t from δ_{t+1} and $\mathbf{x}_t$.
3. **update**: if the layer has parameters, compute the gradient of the loss wrt these parameters from δ_t and $\mathbf{x}_t$.



Linear layer

Parameters: $\{W, \mathbf{b}\}$

Operations:

- forward:

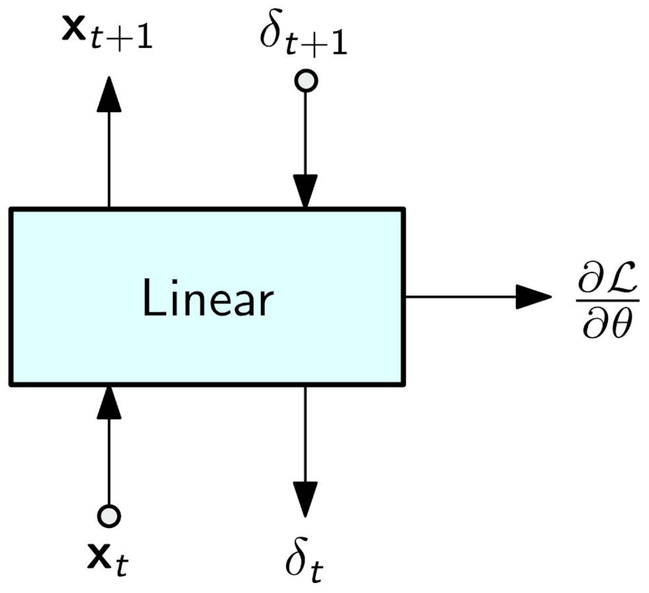
$$\mathbf{x}_{t+1} = W\mathbf{x}_t + \mathbf{b}$$

- backward:

$$\delta_t = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{t+1}} \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} = W^T \delta_{t+1}$$

- update:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{t+1}} \frac{\partial \mathbf{x}_{t+1}}{\partial W} = \delta_{t+1} \mathbf{x}_t^T$$



ReLU layer

Parameters: None

Operations:

- forward:

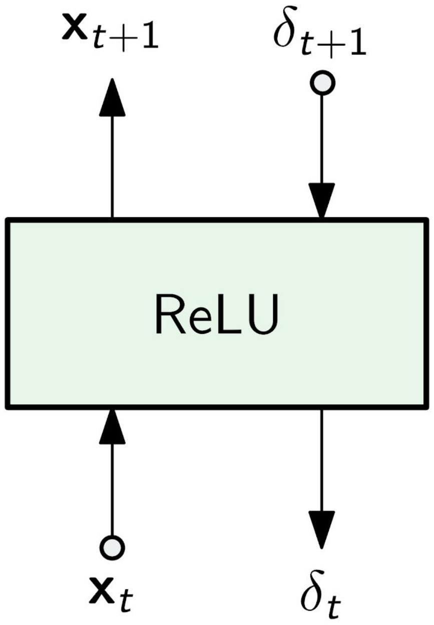
$$\mathbf{x}_{t+1} = r(\mathbf{x}_t)$$

- backward:

$$\delta_t = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{t+1}} \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} = \delta_{t+1} \odot r'(\mathbf{x}_t)$$

With:

$$r(x) = \max\{0, x\} \qquad r'(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

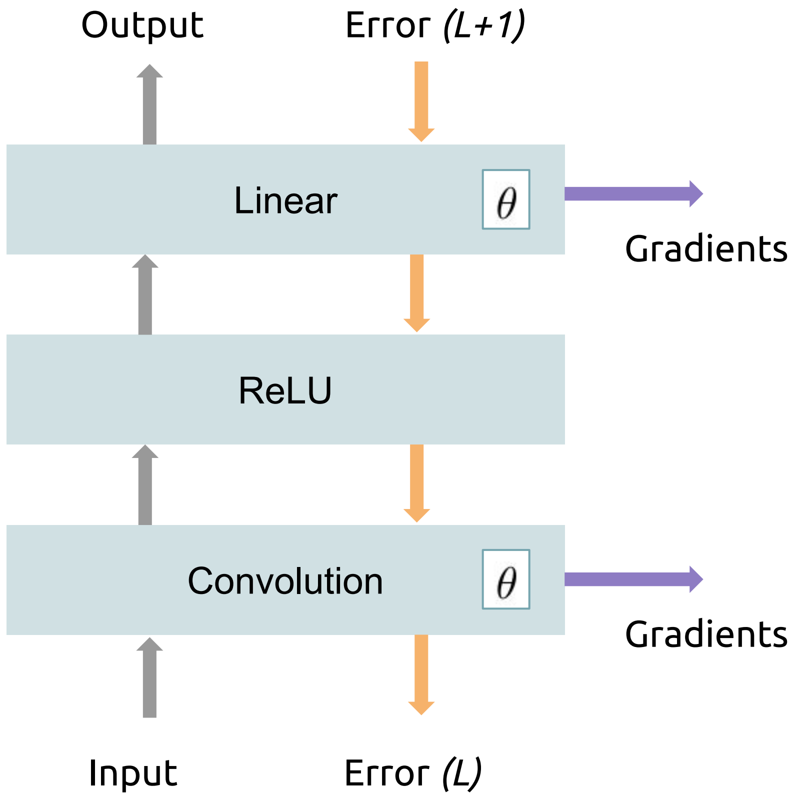


Modular backprop

Using this idea, it is possible to create many types of layers

- Linear (fully connected layers)
- Activation functions (sigmoid, ReLU)
- Convolutions
- Pooling
- Dropout

Once layers support the backward and forward operations, they can be plugged together to create more complex functions



Implementation notes

Caffe and Torch

Libraries like Caffe and Torch implement backpropagation this way.

To define a new layer, you need to create an class and define the **forward** and **backward** operations.

Theano and TensorFlow

Libraries like Theano and TensorFlow operate on a computational graph.

To define a new layer, you only need to specify the **forward** operation. Autodiff is used to automatically infer backward.

You also don't need to implement backprop manually in Theano or TensorFlow. It uses computational graph optimizations to automatically factor out common computations.

Practical tips for training deep nets

Choosing hyperparameters

Can already see we have lots of **hyperparameters** to choose:

1. Learning rate
2. Regularization constant
3. Number of epochs
4. Number of hidden layers
5. Nodes in each hidden layer
6. Weight initialization strategy
7. Loss function
8. Activation functions
9. ...

:(

Choosing these is a bit of an art.

Good news: in practice many configurations work well

There are some reasonable **heuristics**. E.g

1. Try 0.1 for the learning rate. If this diverges, divide by 3. Repeat.
2. Try an existing network architecture and adapt it for your problem
3. Try overfit the data with a big model, then regularize

You can also do a **hyperparameter search** if you have enough compute:

- Randomized search tends to work well

Choosing the learning rate

For **first order** optimization methods, we need to choose a learning rate (aka **step size**)

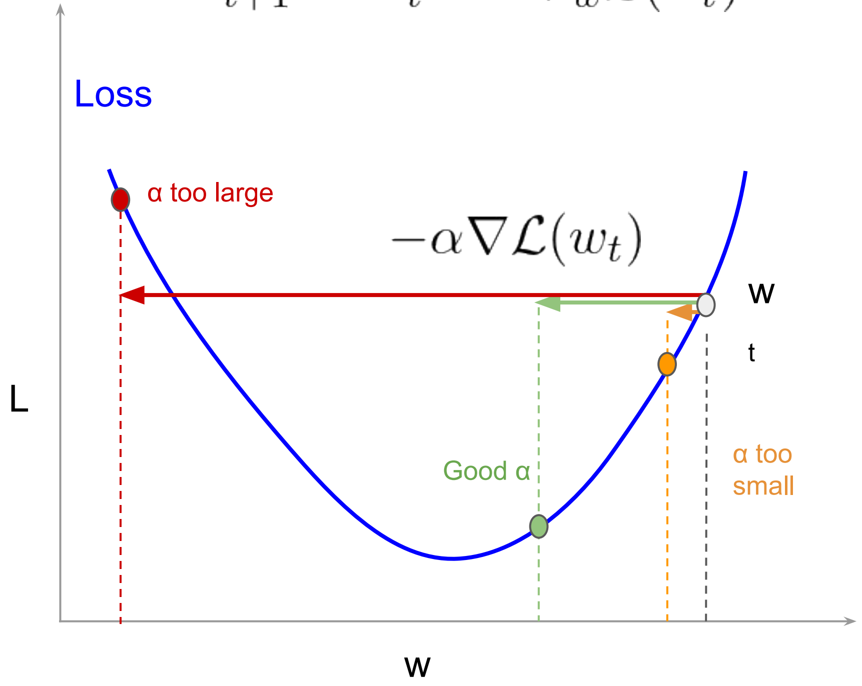
- **Too large**: overshoots local minimum, loss increases
- **Too small**: makes very slow progress, can get stuck
- **Good learning rate**: makes steady progress toward local minimum

Usually want a higher learning rate at the start and a lower one later on.

Common strategy in practice:

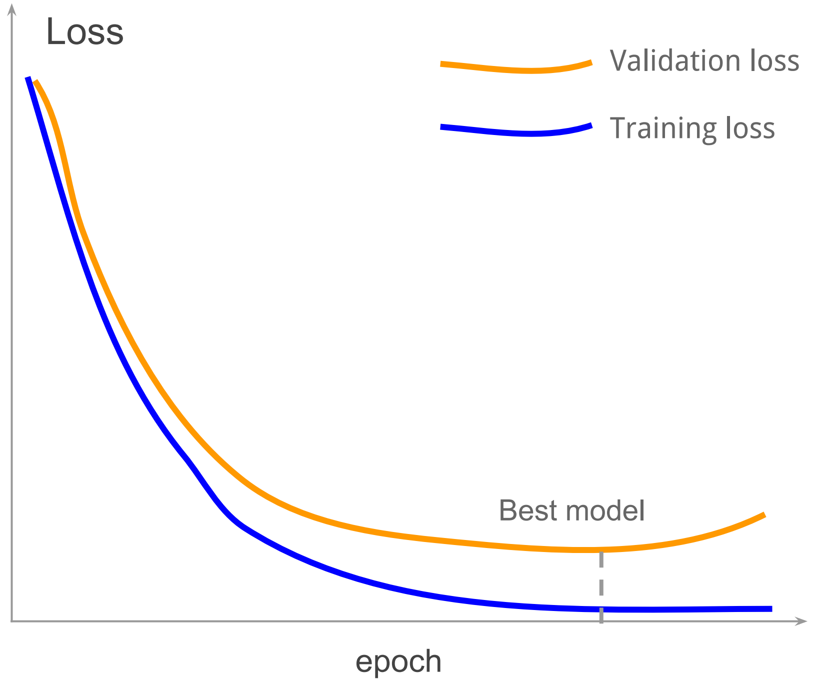
- Start off with a high LR (like 0.1 - 0.001),
- Run for several **epochs** (1 - 10)
- Decrease LR by multiplying a constant factor (0.1 - 0.5)

$$w_{t+1} = w_t - \alpha \nabla_w \mathcal{L}(w_t)$$



Training and monitoring progress

1. Split data into train, validation, and test sets
    - Keep 5-30% of data for validation
2. Fit model parameters on train set using SGD
3. After each epoch:
    - **Test model on validation set** and compute loss
        - Also compute whatever other metrics you are interested in, e.g. top-5 accuracy
    - Save a snapshot of the model
4. Plot **learning curves** as training progresses
5. Stop when validation loss starts to increase
6. Use model with minimum validation loss



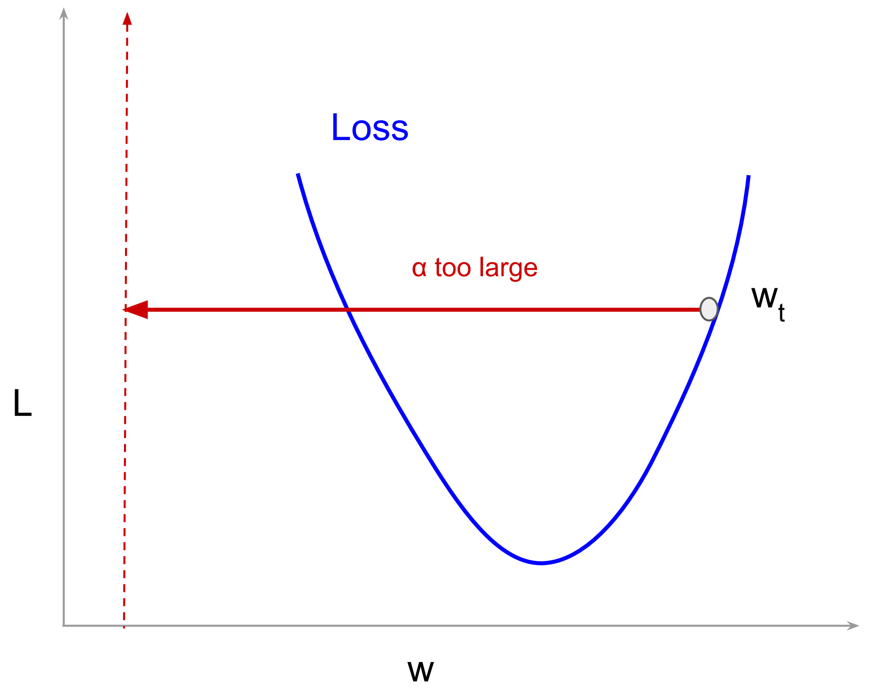
Divergence

Symptoms:

- Training loss keeps increasing
- Inf, NaN in loss

Try:

- Reduce learning rate
- Zero center and scale inputs/targets
- Check weight initialization strategy (monitor gradients)
- Numerically check your gradients
- Clip gradient norm



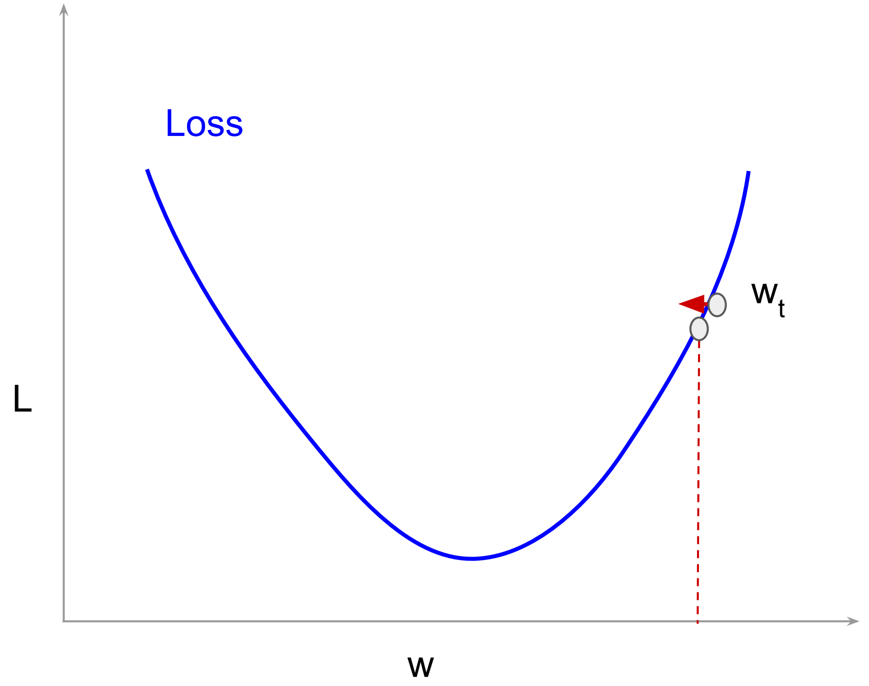
Slow convergence

Symptoms:

- Training loss decreases slowly
- Training loss does not decrease

Try:

- Increase learning rate
- Zero center and scale inputs/targets
- Check weight initialization strategy (monitor gradients)
- Numerically check gradients
- Use ReLUs
- Change batch size (careful)
- Change loss, model architecture



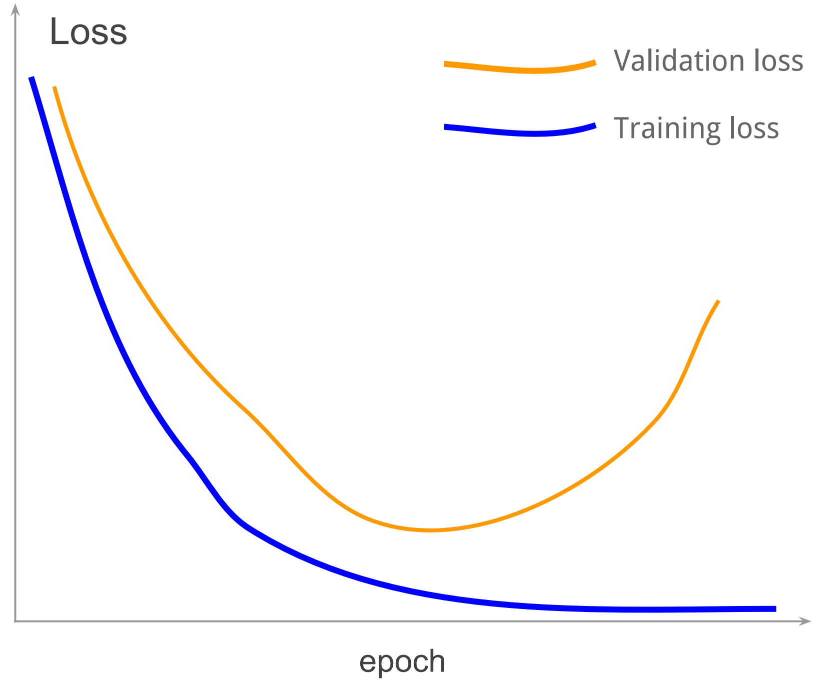
Overfitting

Symptoms:

- Validation loss decreases at first, then starts increasing
- Training loss continues to go down

Try:

- Find more training data
- Add stronger regularization
  - dropout, drop-connect, L^2
- Data augmentation (flips, rotations, noise)
- Reduce complexity of your model



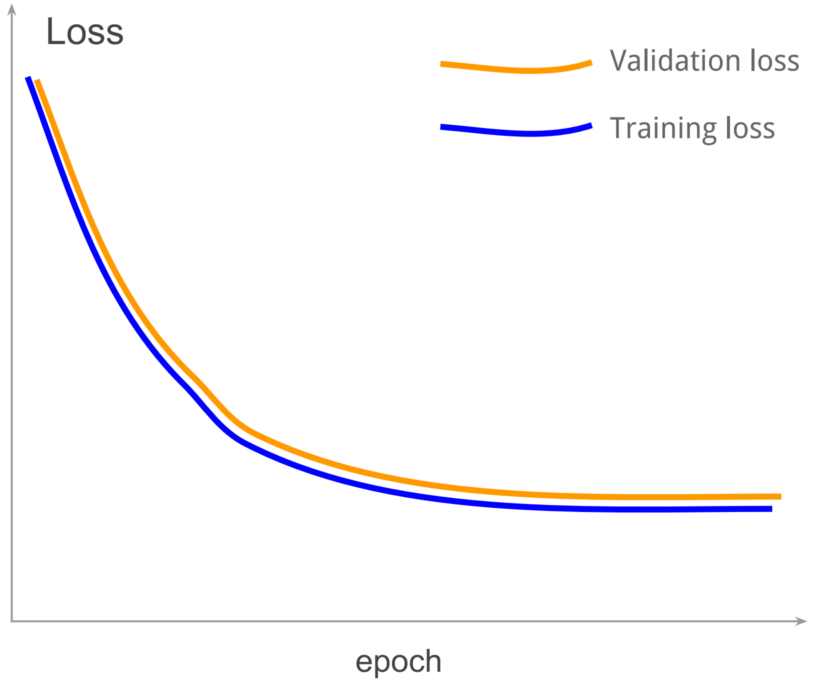
Underfitting

Symptoms:

- Training loss decreases at first but then stops
- Training loss still high
- Training loss tracks validation loss

Try:

- Increase model capacity
  - Add more layers, increase layer size
- Use more suitable network architecture
  - E.g. multi-scale architecture
- Decrease regularization strength



Questions?