Deep Learning

Deep Neural Network

Lionel Fillatre

2024-2025

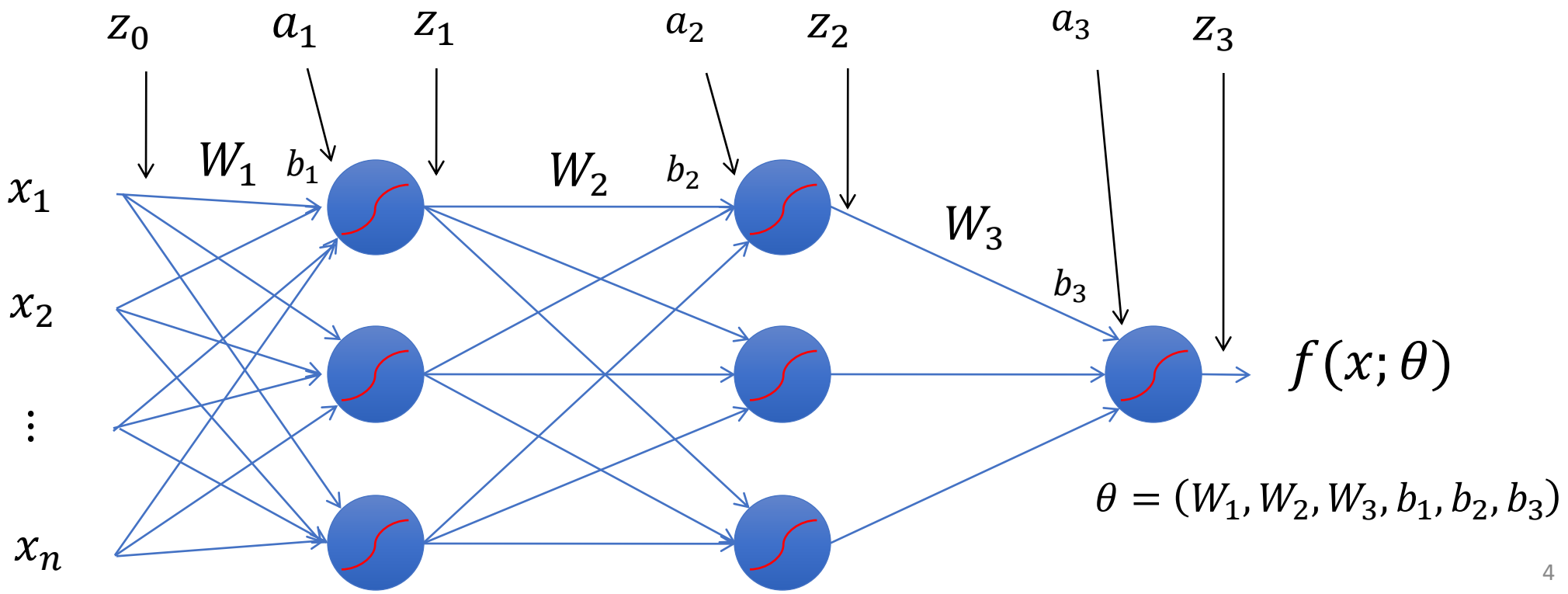
Outline of Lecture 2

- Deep Neural Network
- Gradient descent
- Automatic differentiation
- Backpropagation
- Weight Initialization
- Conclusion

Deep Neural Network

Feed-Forward Networks and All Its Variables

- Predictions are fed forward through the network to classify:
  - $z_0 = x$
  - $a_k = W_k z_{k-1} + b_k,\ k = 1, 2, 3$
  - $z_k = \sigma(a_k),\ k = 1, 2, 3$
  - $f(x) = f(x; \theta) = z_3$



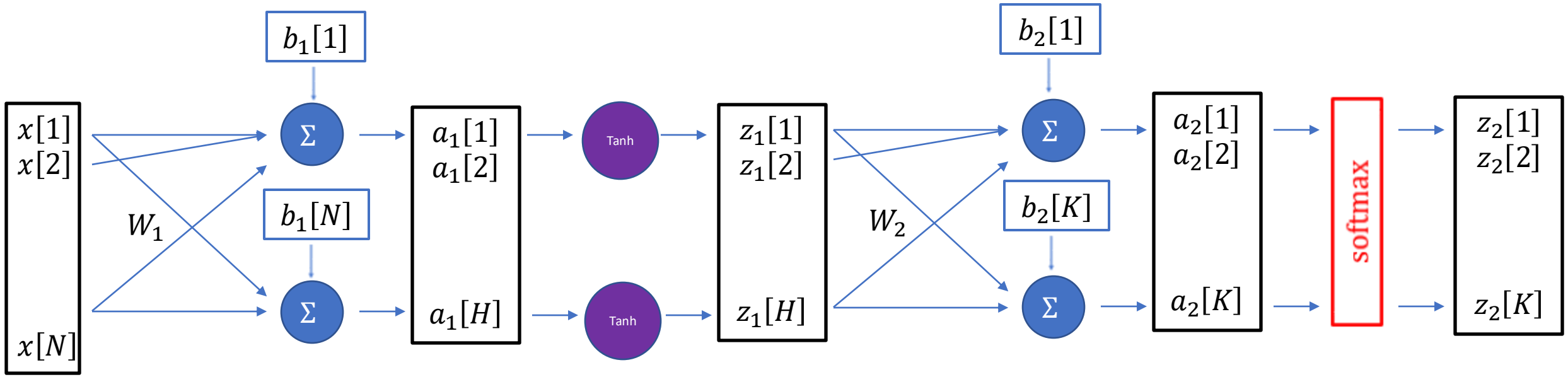
Terminology

- A feedforward net is a nonlinear function composed of repeated affine transformation followed by a nonlinear action:

$$\begin{cases} f_0(x) = x \\ f_k(x) = \sigma_k(W_k f_{k-1}(x) + b_k), & 1 \leq k \leq L-1 \\ \hat{y} = f(x) = f_L(x) = \sigma_L(W_L f_{L-1}(x) + b_L). \end{cases}$$

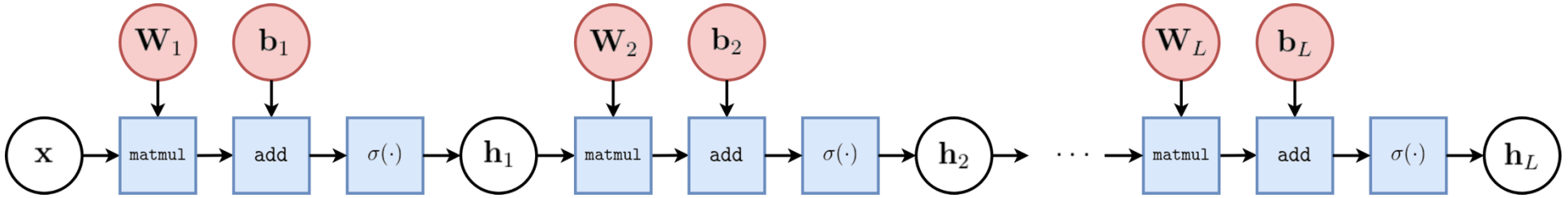
- We say that the netwoks contains L layers and $L-1$ hidden layers
- The size of the layers, $n_0, n_1, \ldots, n_L$, corresponds to the number of neurons
- The weights of layer k are given by matrix $W_k \in \mathbb{R}^{n_k \times n_{k-1}}$
- The biases of layer k are given by $b_k \in \mathbb{R}^{n_k}$
- The nonlinear activation function of layer k is $\sigma_k(\quad)$

Pytorch Implementation




```
model = torch.nn.Sequential(
        torch.nn.Linear(N, H), # weight matrix dim [N x H]
        torch.nn.Tanh(),
        torch.nn.Linear(H, K), # weight matrix dim [H x K]
        torch.nn.Softmax(),
)
```

Computational graph view



Gradient Descent

Reminder: Learning Criterion

- We observe some samples $\mathcal{D} = (x_i, y_i)_{i=1,\cdots,N}$ following the distribution of (X, Y)
  - We are assuming that the samples are independent (i.i.d. assumption)

- We are considering a family $\mathcal{F}$ of functions f_θ parameterized by θ
  - The parameter θ generally belongs to an Euclidean space Θ (e.g., $\theta \in \mathbb{R}^n$)

- We are expecting to solve

$$\theta^* \in \underset{\theta \in \Theta}{\operatorname{argmin}} \, \mathbb{E}\big(\ell(f_\theta(X), Y)\big)$$

but, in practice, a naïve approach consists in minimizing the empirical risk

$$\hat{\theta} = \hat{\theta}(\mathcal{D}) \in \underset{\theta \in \Theta}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

Optimization algorithms

Gradient Descent over $\mathcal{D} = (x_i, y_i)_{i=1,\cdots,N}$

- Require: Sequence of learning rates $\alpha^{(k)}$
- Require: Initial Parameter $\theta^{(0)}$
- Algorithm:

while stopping criteria not met **do**

- Compute steepest descent estimate over N examples $\left(x_{(i)}, y_{(i)}\right)$:

$$g^{(k)} = -\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \ell\left(f\left(x_{(i)}; \theta^{(k)}\right), y_{(i)}\right)$$

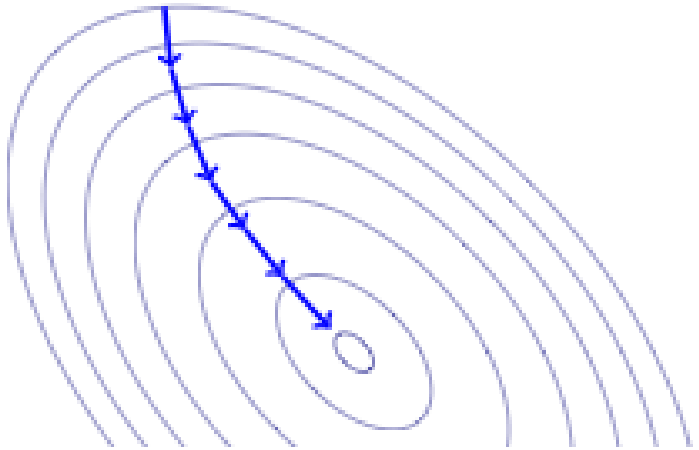
- Apply Update: $\theta^{(k)} + \alpha^{(k)}\ g^{(k)}$

end while

Gradient Descent

- Positive:
  - Gradient estimates are stable if N is large

- Negative:
  - Need to compute gradients over the entire training for one update



Stochastic Gradient Descent

- Require: Sequence of learning rates $\alpha^{(k)}$
- Require: Initial Parameter $\theta^{(0)}$
- Algorithm:

while stopping criteria not met **do**

- Sample over $m \ll N$ examples $\left(x_{(i)}, y_{(i)}\right)$ (this is called a **mini-batch or a batch**):

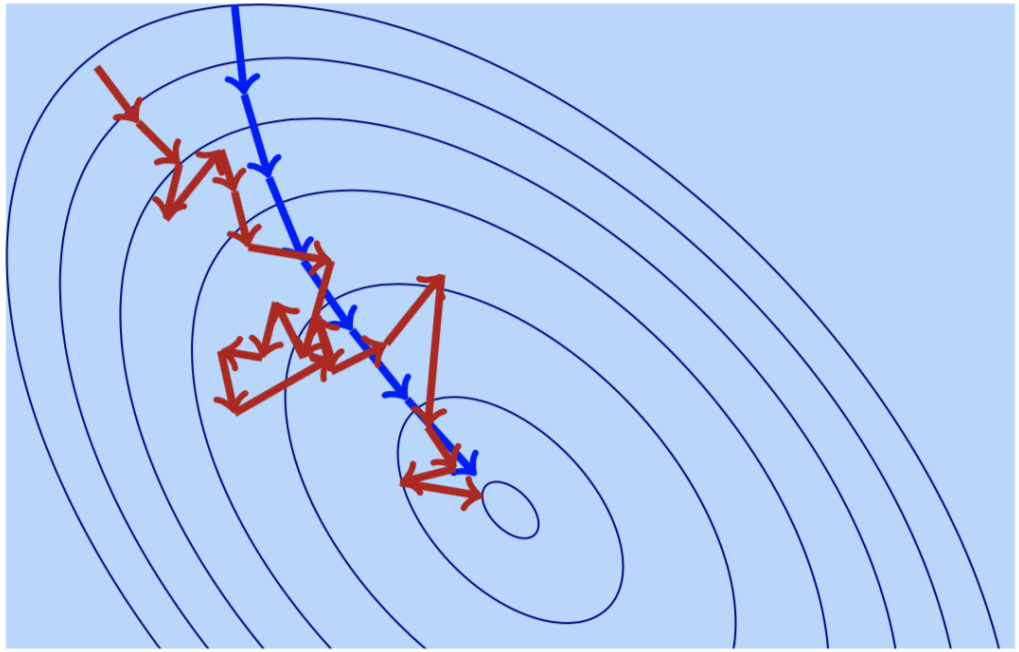
$$g^{(k)} = -\frac{1}{m}\sum_{i=1}^{m} \nabla_{\theta} \ell\left(f\left(x_{(i)}; \theta^{(k)}\right), y_{(i)}\right)$$

- Apply Update: $\theta^{(k)} + \alpha^{(k)}\, g^{(k)}$

end while

Batching

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger batches
- Advantage: Computation time per update does not depend on number of training examples N
- This allows convergence on extremely large datasets



Stochastic Gradient Descent

- Sufficient condition to guarantee convergence:
  - $\sum_{k=1}^{\infty} \alpha^{(k)} = \infty$
  - $\sum_{k=1}^{\infty} {\alpha^{(k)}}^2 < \infty$

- In practice, many ways to fix the learning rate
  - Time-based learning rate:

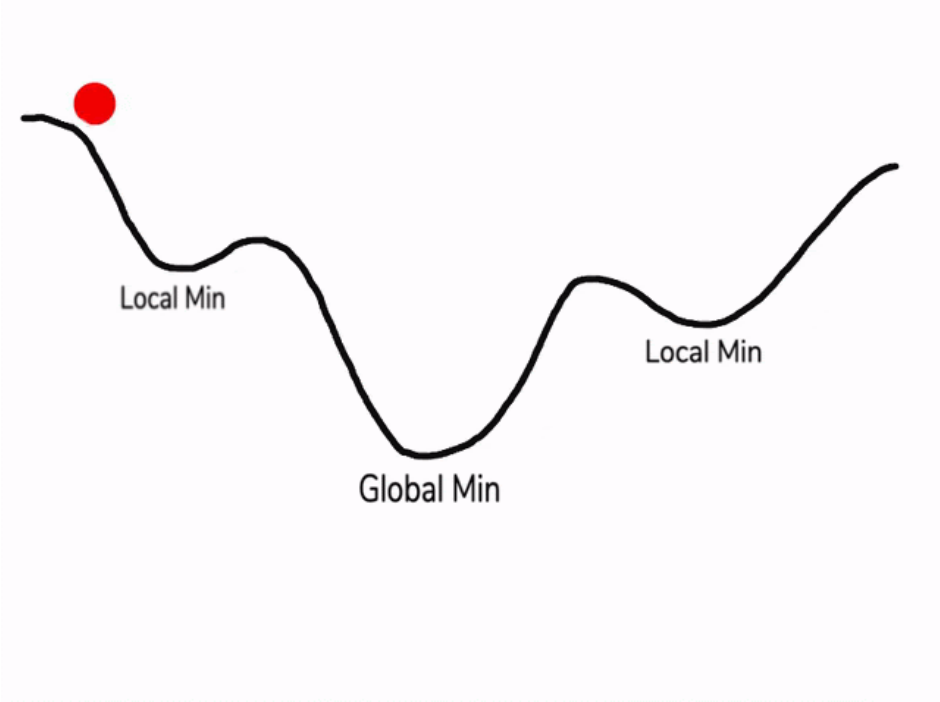
$$\alpha^{(k)} = \frac{\alpha^{(k-1)}}{1 + \tau k},$$

  - Exponential learning rate:

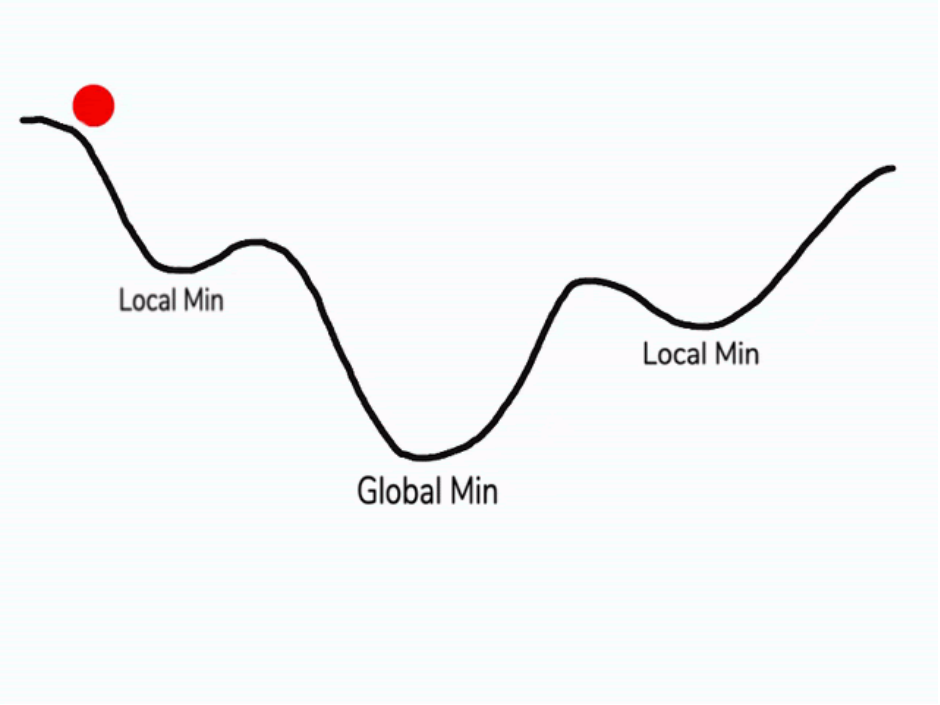
$$\alpha^{(k)} = \alpha^{(0)} e^{-\tau k}$$

  where τ is the decay and $\alpha^{(0)}$ is chosen conveniently

Momentum: why?



Without momentum



With momentum

https://medium.com/@kaitotally/adam-the-birthchild-of-adagrad-and-rmsprop-b5308b24b9cd

Momentum

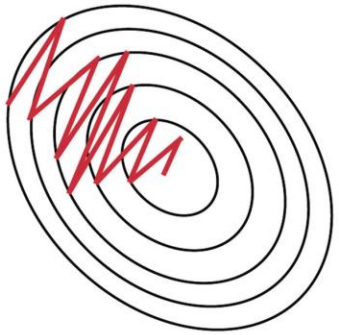
- Momentum technique is an approach which provides an update rule that is motivated from a physical perspective.
- Imagine a ball in a hilly terrain is trying to reach the deepest valley.
  - When the slope of the hill is very high, the ball gains a lot of momentum and is able to pass through slight hills in its way.
  - As the slope decreases the momentum and speed of the ball decreases, eventually coming to rest in the deepest position of valley.
- Unlike in classical stochastic gradient descent, momentum technique tends to keep traveling in the same direction, preventing oscillations.

- Algorithm:

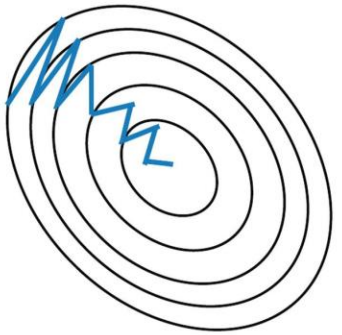
while stopping criteria not met **do**

- Compute the steepest descent $g^{(k)}$
- Compute $d^{(k)} = \mu d^{(k-1)} + \alpha g^{(k)}$
- Apply Update: $\theta^{(k)} + d^{(k)}$

end while



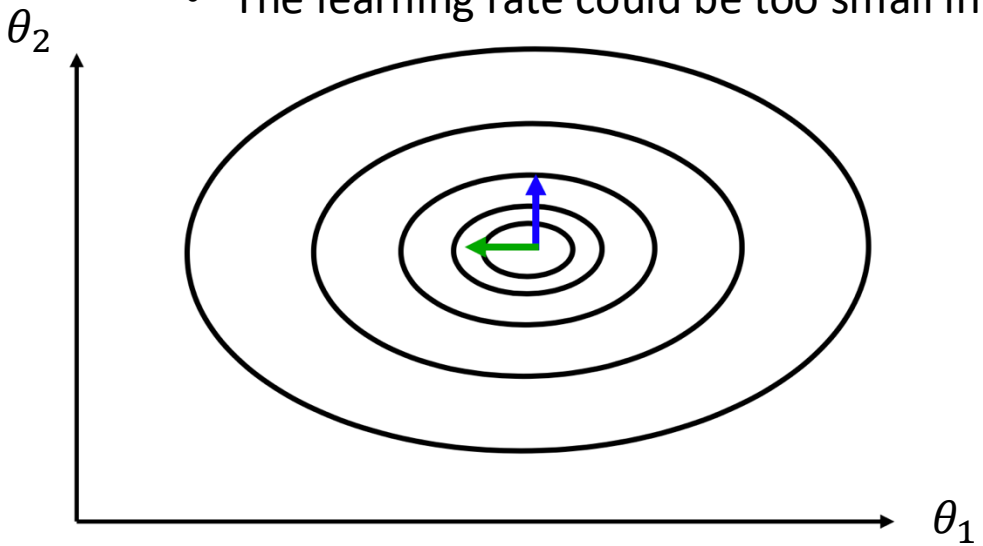
Stochastic Gradient Descent **withhout** Momentum

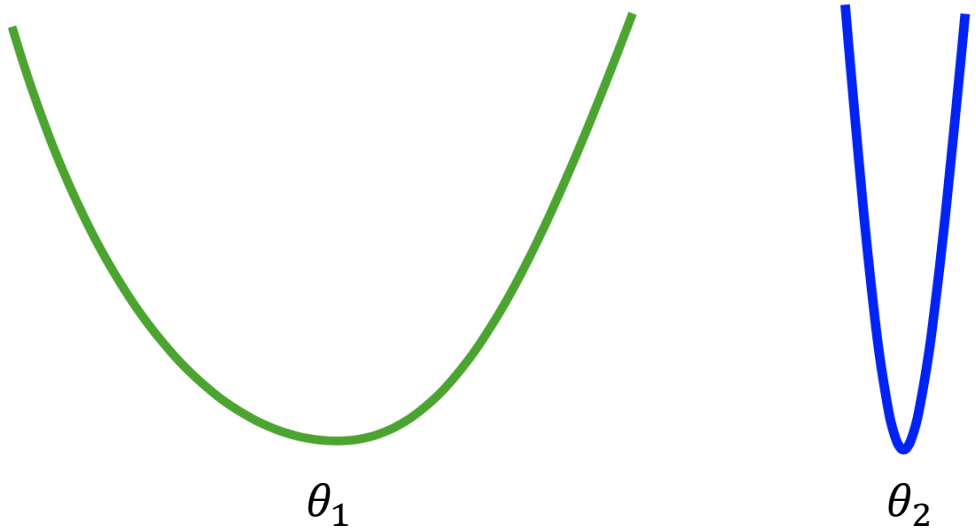


Stochastic Gradient Descent **with** Momentum

Adagrad

- The learning rate might be very difficult to set
  - if we set it too small, then the parameter update will be very slow and it will take very long time to achieve an acceptable loss.
  - Otherwise, if we set it too large, then the parameter will move all over the function and may never achieve acceptable loss at all.
- To make things worse, the high-dimensional non-convex nature of neural networks optimization could lead to different sensitivity on each dimension.
  - The learning rate could be too small in some dimension and could be too large in another dimension.





Adagrad

- **Principle:**
  - When the previous updates of $\theta_i^{(k)}$ are small, we may augment the next update
  - When the previous updates of $\theta_i^{(k)}$ are large, we may decrease the next update

- **Method:**
  - Scale the descent with the square root of the sum of the gradients
  - Use a small constant $\delta > 0$ for numerical stability
- **Algorithm:**

while stopping criteria not met **do**

- Compute the steepest descent $g^{(k)}$
- Accumulate $r^{(k)} = r^{(k-1)} + g^{(k)} \odot g^{(k)}$
- Compute $d^{(k)} = \frac{1}{\delta + \sqrt{r^{(k)}}} \odot g^{(k)}$
- Apply Update: $\theta^{(k)} + \alpha d^{(k)}$

end while

$$r^{(0)} = 0$$

Hadamard product:

$$\begin{pmatrix} a_1 \\ \vdots \\ a_d \end{pmatrix} \odot \begin{pmatrix} b_1 \\ \vdots \\ b_d \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ \vdots \\ a_d b_d \end{pmatrix}$$

Final Comments

- A plenty of other algorithms: RMSProp, RMSProp with Nesterov, Adam

- Many hyperparameters to choose
  - Network size/depth
  - Model variations (activation functions, etc.)
  - Minibatch creation strategy
  - Optimizer/learning rate

- Full models are complicated and opaque, debugging can be difficult!

Gradient

- To minimize $\mathcal{L}(\theta)$ with gradient descent, we need the gradient $\nabla_\theta \mathcal{L}(\theta)$

$$\mathcal{L}(\theta) = \sum_{x_i, y_i} \ell(f(x_i; \theta), y_i)$$

- Therefore, it requires the evaluation of the (total) derivatives

$$\frac{\partial \ell}{\partial W_{i,j}^{(k)}}, \frac{\partial \ell}{\partial b_i^{(k)}}$$

of the loss $\ell(f(x; \theta), y)$ with respect to all model parameters $W_{i,j}^{(k)}$, $b_i^{(k)}$ for all i, j and all layer k.

- These derivatives can be evaluated automatically from the computational graph of $\mathcal{L}(\theta)$ using backpropagation / automatic differentiation.

Automatic Differentiation

Analytical computation

- All derivatives we care about are the derivatives of the error function with respect to the model parameters:
  - The error function is scalar
  - We need its gradient

- We are going to write the error function as a composition of simpler functions, and use the chain rule to compute the gradient efficiently

- The error function can be as complex as a program with control flow statements

- Each component function, called a unit, is assumed to be at least piecewise differentiable with a known formula for its derivative

Chain rule

- The most important concept for backpropagation is the **chain rule**

- **Chain rule:**
  - Let a function $f: \mathbb{R} \to \mathbb{R}$ defined by

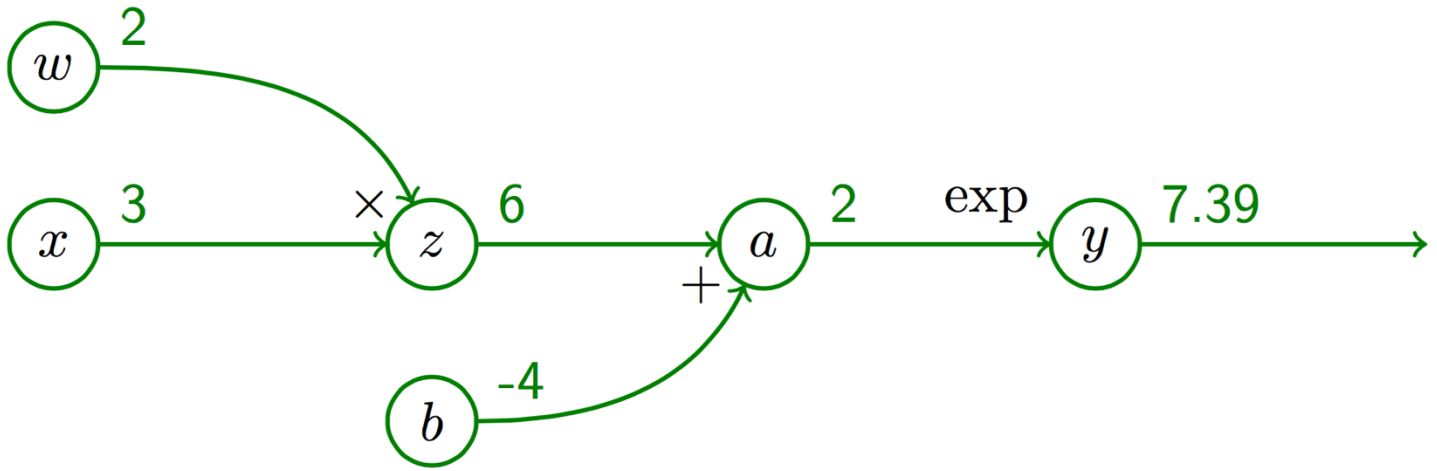
$$f(x) = h(g(x)) = h \circ g(x), \forall x \in \mathbb{R}$$

  with $g: \mathbb{R} \to \mathbb{R}$ and $h: \mathbb{R} \to \mathbb{R}$ differentiable

  - Then, for any real a,

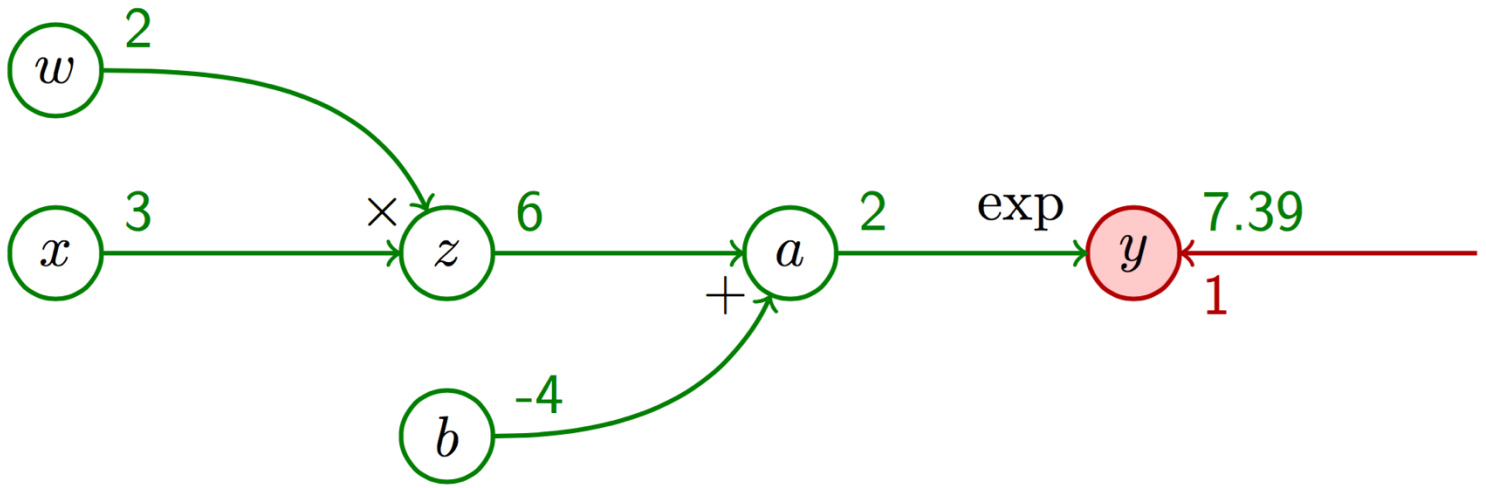
$$\frac{\partial f}{\partial x}(a) = \frac{\partial h}{\partial x}(g(a)) \times \frac{\partial g}{\partial x}(a)$$

Example



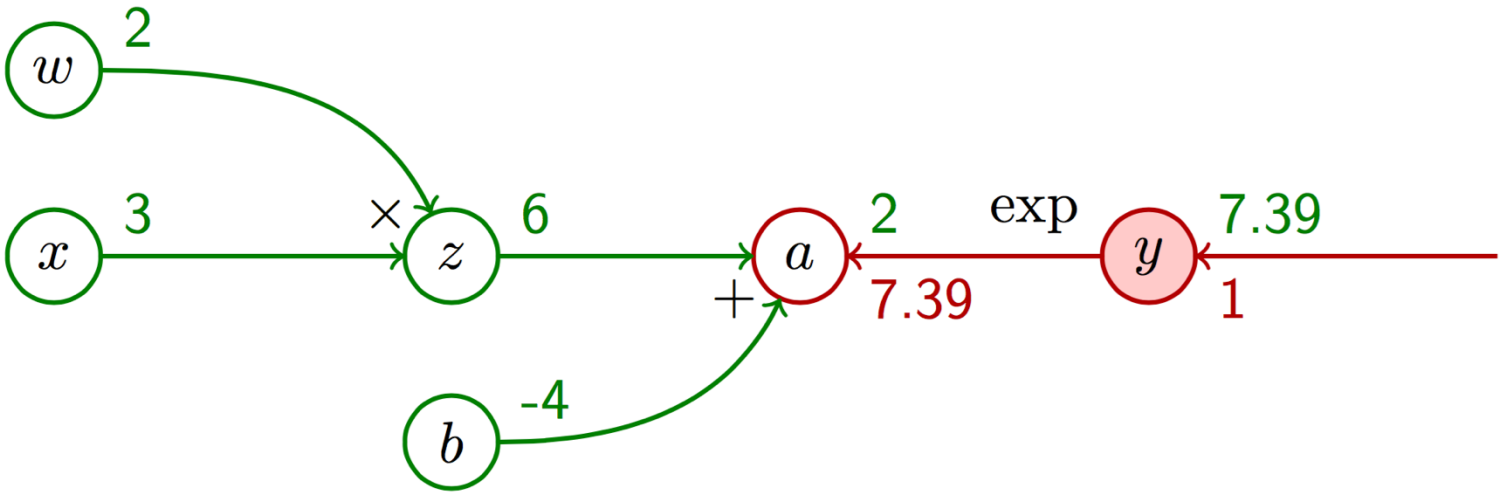
- $y = e^{wx+b}$ is broken down into $y = e^{a}$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$

Example



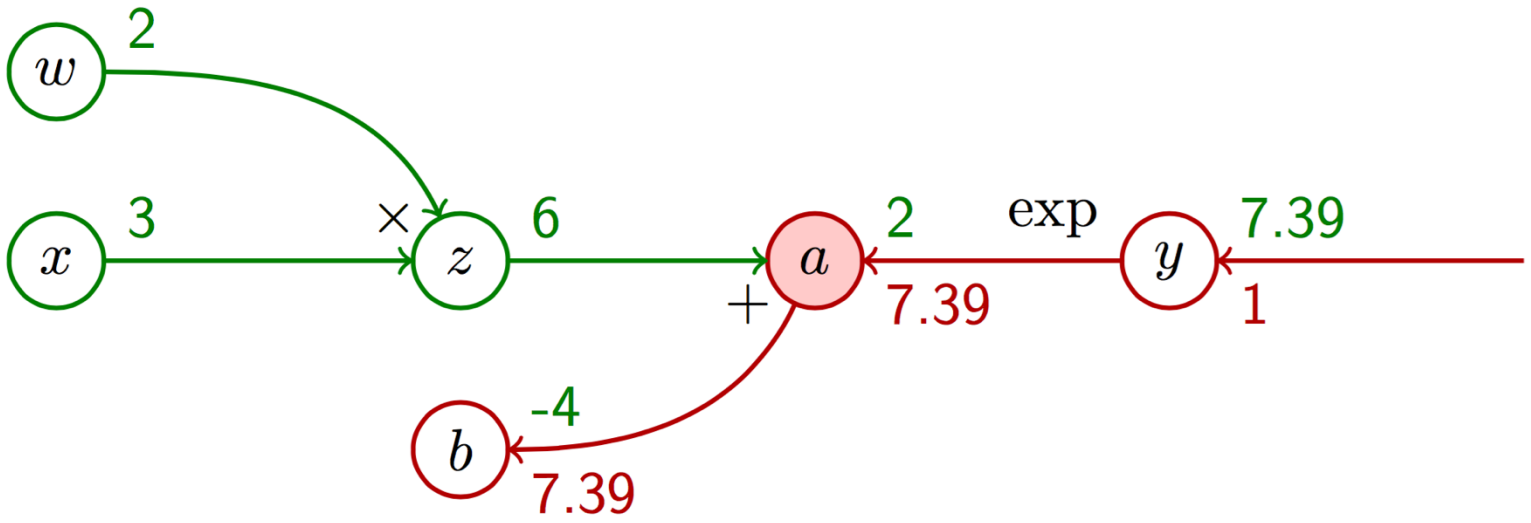
- $y = e^{wx+b}$ is broken down into $y = e^{a}$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1$

Example



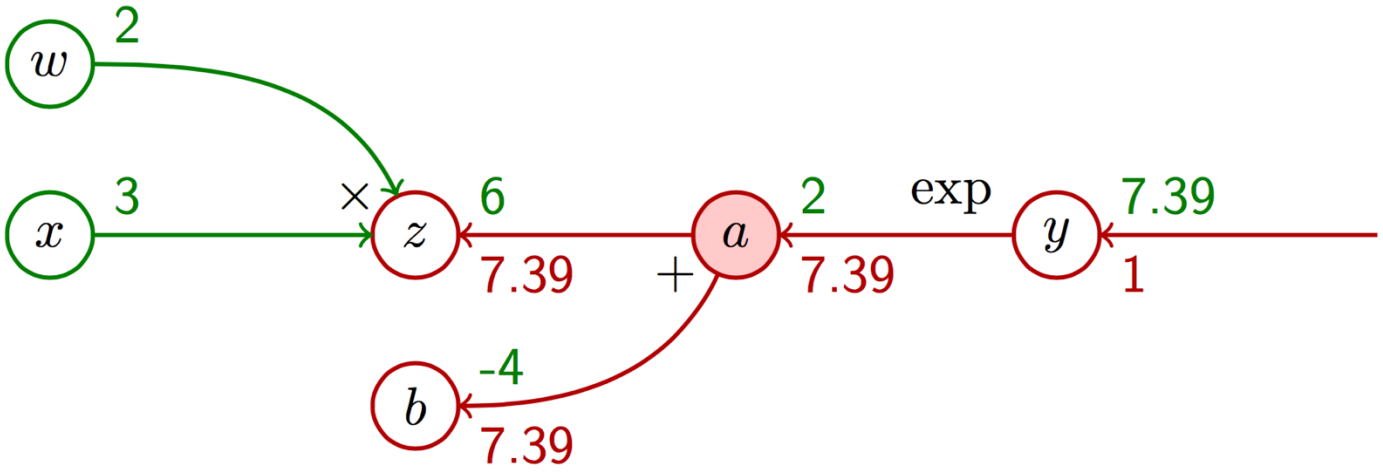
- $y = e^{wx+b}$ is broken down into $y = e^a$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1, \frac{\partial y}{\partial a} = e^a = 7.39$

Example



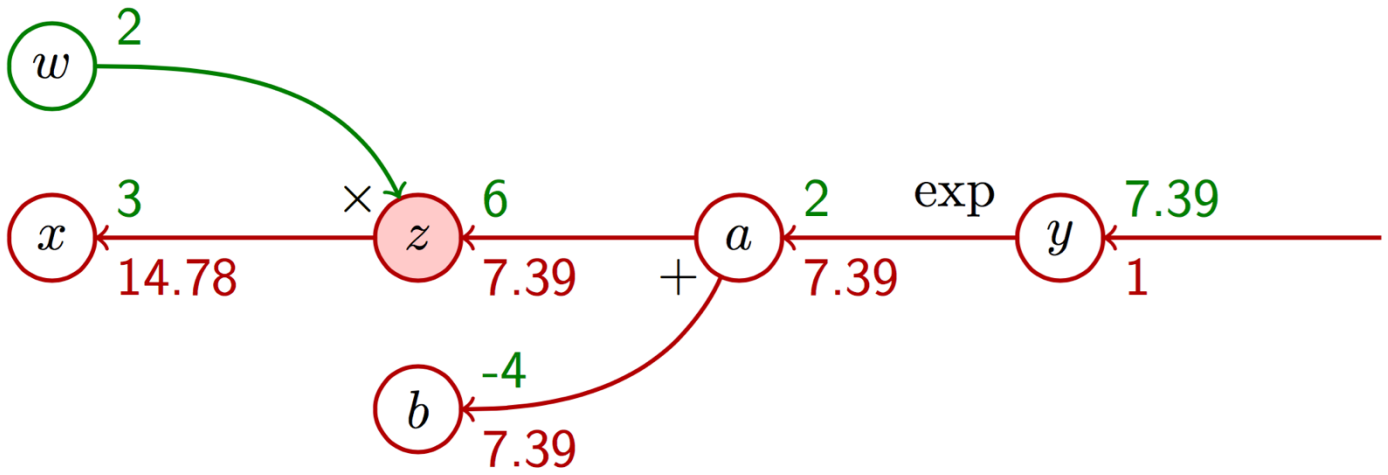
- $y = e^{wx+b}$ is broken down into $y = e^a$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1, \frac{\partial y}{\partial a} = e^a = 7.39, \frac{\partial y}{\partial b} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial b} = \frac{\partial y}{\partial a} = 7.39$

Example



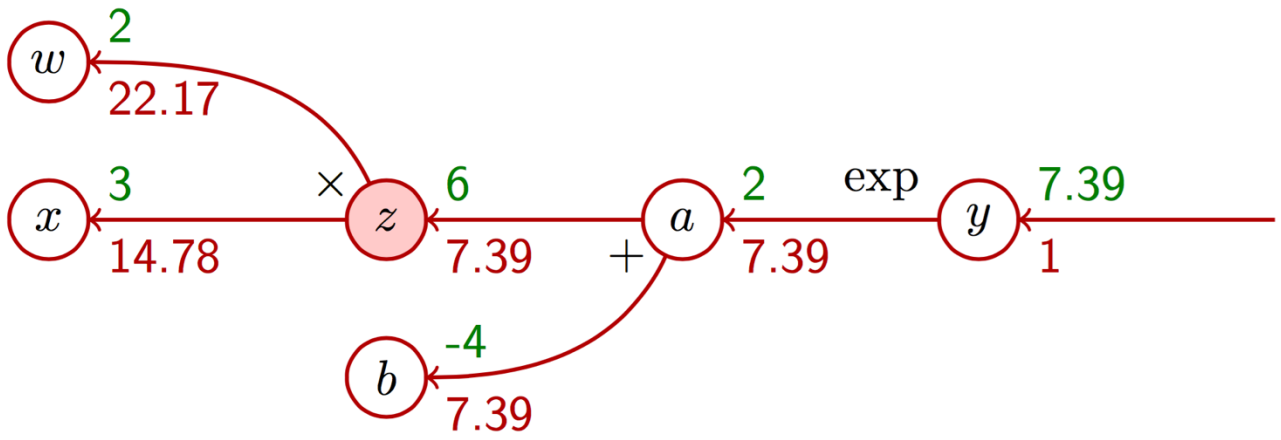
- $y = e^{wx+b}$ is broken down into $y = e^a$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1, \frac{\partial y}{\partial a} = e^a = 7.39, \frac{\partial y}{\partial b} = \frac{\partial y}{\partial a}\frac{\partial y}{\partial b} = \frac{\partial y}{\partial a} = 7.39, \frac{\partial y}{\partial z} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial y}{\partial a} = 7.39$

Example



- $y = e^{wx+b}$ is broken down into $y = e^a$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1, \frac{\partial y}{\partial a} = e^a = 7.39, \frac{\partial y}{\partial b} = \frac{\partial y}{\partial a}\frac{\partial y}{\partial b} = \frac{\partial y}{\partial a} = 7.39, \frac{\partial y}{\partial z} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial y}{\partial a} = 7.39$
- $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial y}{\partial z}w = 14.78$

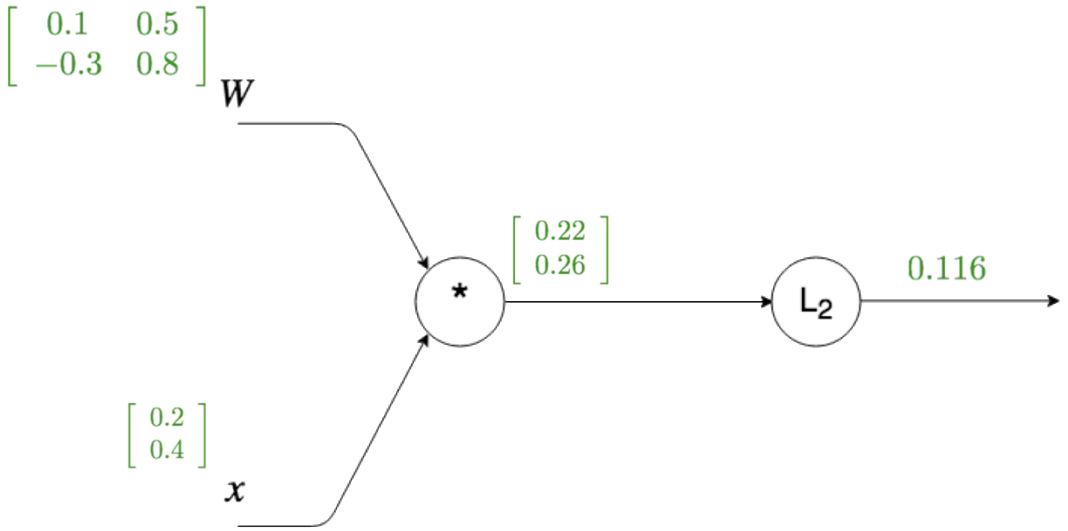
Example



- $y = e^{wx+b}$ is broken down into $y = e^a$, $a = z + b$, $z = wx$
- We seek $\frac{\partial y}{\partial w}, \frac{\partial y}{\partial x}, \frac{\partial y}{\partial b}$ for $(w, x, b) = (2,3,-4)$
- $\frac{\partial y}{\partial y} = 1, \frac{\partial y}{\partial a} = e^a = 7.39, \frac{\partial y}{\partial b} = \frac{\partial y}{\partial a}\frac{\partial y}{\partial b} = \frac{\partial y}{\partial a} = 7.39, \frac{\partial y}{\partial z} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial a}{\partial z} = 7.39$
- $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial y}{\partial z}w = 14.78, \frac{\partial y}{\partial w} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial w} = \frac{\partial y}{\partial z}x = 22.17$

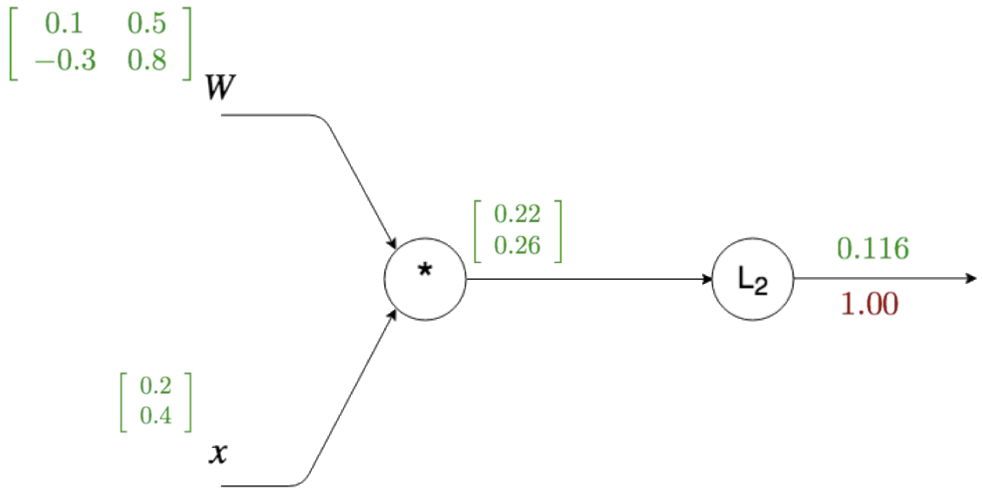
Vectorized example

- $f(x, W) = \|Wx\|^2 = \sum_{i=1}^{n} (Wx)_i^2$
- $x \in \mathbb{R}^n, W \in \mathbb{R}^{n \times n}$
- $y = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$
- $f(x, W) = \|y\|^2 = f(y) = y_1^2 + \cdots + y_n^2$



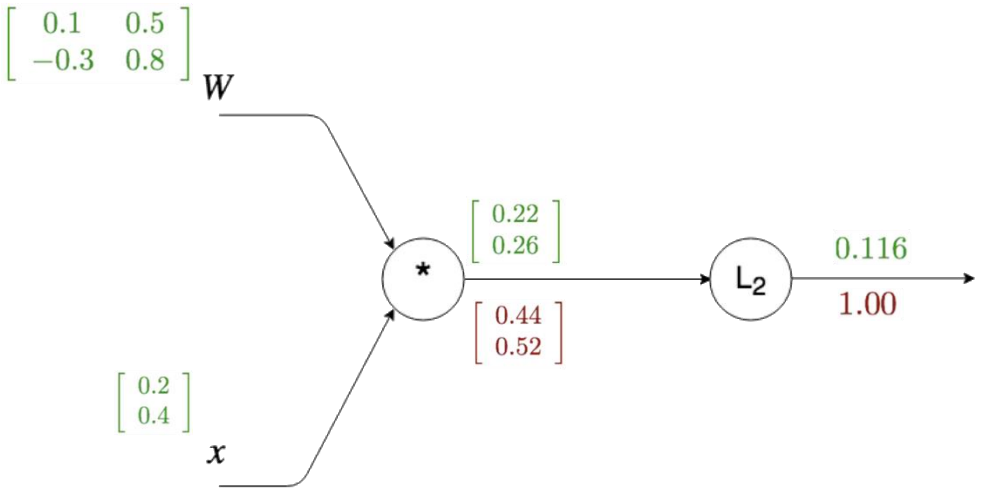
Vectorized example

- $f(x, W) = \|Wx\|^2 = \sum_{i=1}^{n} (Wx)_i^2$
- $y = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$
- $f(x, W) = z = \|y\|^2 = f(y) = y_1^2 + \cdots + y_n^2$
- $\frac{\partial f}{\partial z} = 1$



Vectorized example

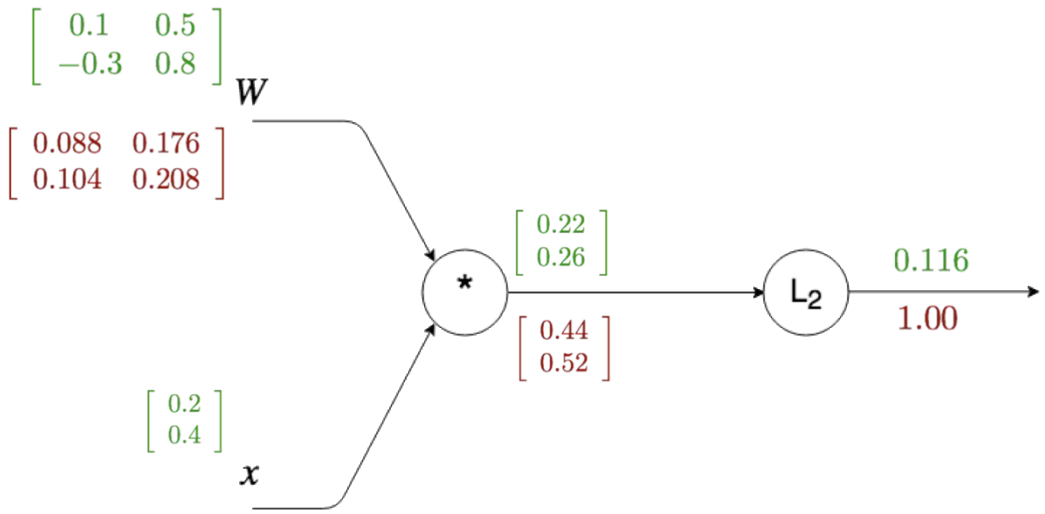
- $f(x, W) = \|Wx\|^2 = \sum_{i=1}^{n} (Wx)_i^2$
- $y = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$
- $f(x, W) = \|y\|^2 = f(y) = y_1^2 + \cdots + y_n^2$
- $\frac{\partial f}{\partial y_i} = 2y_i \Rightarrow \nabla_y f = 2y$



Vectorized example

- $f(x, W) = \|Wx\|^2 = \sum_{i=1}^{n} (Wx)_i^2$
- $y = Wx = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$
- $f(x, W) = \|y\|^2 = f(y) = y_1^2 + \cdots + y_n^2$
- $\frac{\partial f}{\partial y_i} = 2y_i$
- $\frac{\partial y_k}{\partial W_{i,j}} = 1_{\{k=i\}} x_j$
- $\frac{\partial f}{\partial W_{i,j}} = \sum_{k=1}^{n} \frac{\partial f}{\partial y_k} \frac{\partial y_k}{\partial W_{i,j}} = \sum_{k=1}^{n} 2y_k 1_{\{k=i\}} x_j = 2y_k x_j$

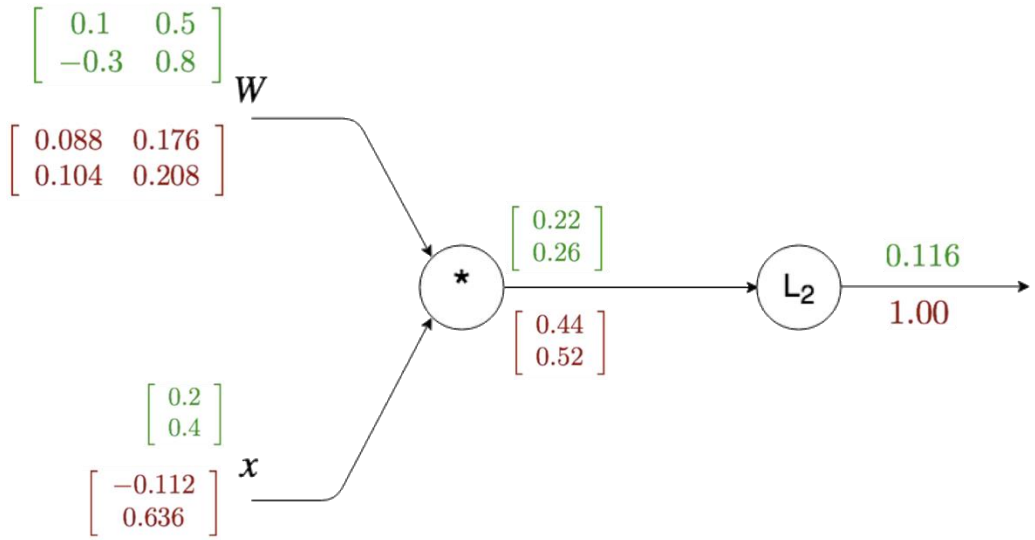
$\Rightarrow \nabla_W f = 2yx^T = 2Wxx^T$



Vectorized example

- $f(x, W) = \|Wx\|^2 = \sum_{i=1}^{n} (Wx)_i^2$
- $y_k = W_{k,1}x_1 + \cdots + W_{k,n}x_n$
- $f(x, W) = \|y\|^2 = f(y) = y_1^2 + \cdots + y_n^2$
- $\frac{\partial f}{\partial y_i} = 2y_i$
- $\frac{\partial y_k}{\partial x_i} = W_{k,i}$
- $\frac{\partial f}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial f}{\partial y_k} \frac{\partial y_k}{\partial x_i} = \sum_{k=1}^{n} 2y_k W_{k,i}$

$\Rightarrow \nabla_x f = 2W^T y = 2W^T Wx$



Backpropagation

Backpropagation/Automatic differentiation

- Consider a 1-dimensional output composition $f \circ g$, such that

$$y = f(u)$$
$$u = g(x) = \big(g_1(x), \dots, g_m(x)\big) \in \mathbb{R}^m$$

- The chain rule of total derivatives states that

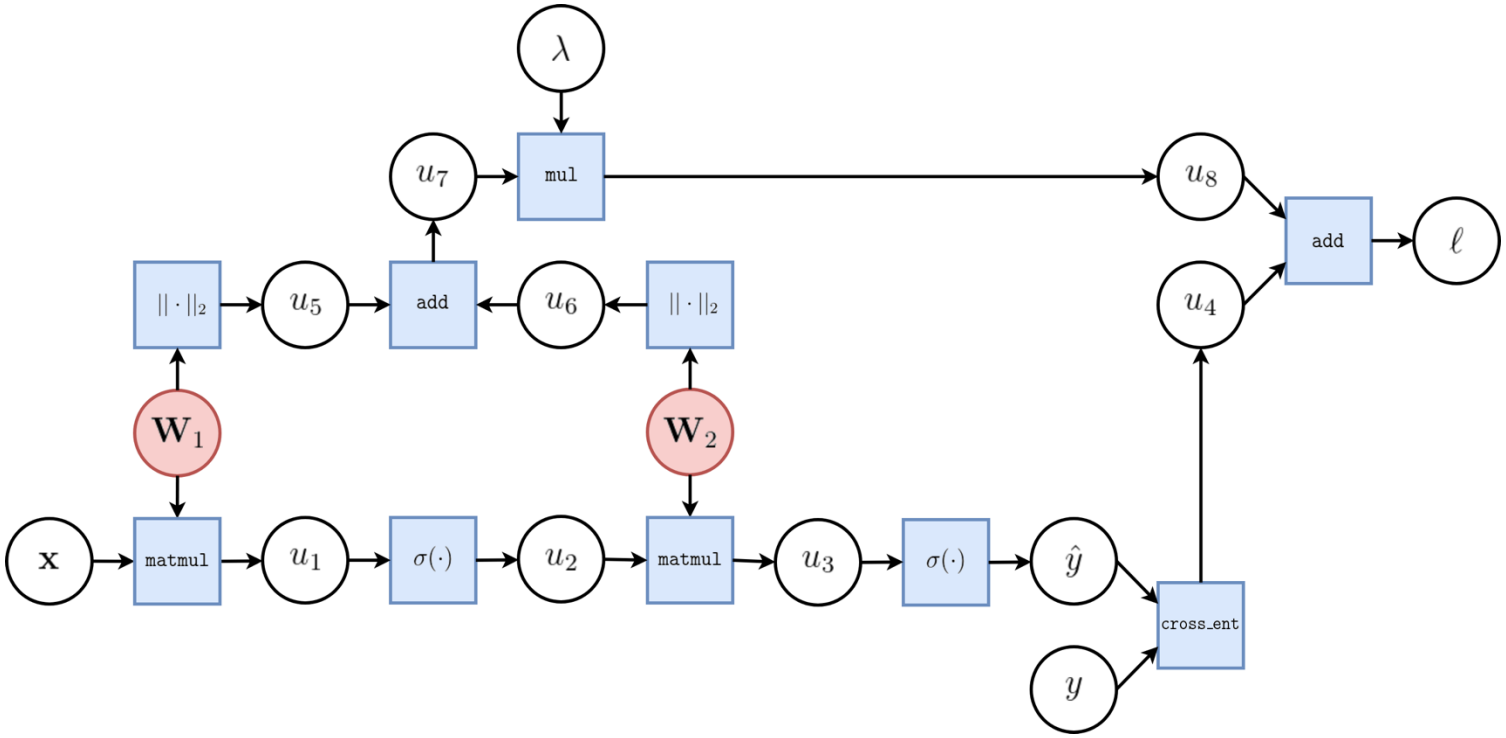
$$\frac{dy}{dx} = \sum_{k=1}^{n} \frac{\partial y}{\partial u_k} \times \underbrace{\frac{du_k}{dx}}_{\text{recursive case}}$$

- Since a neural network is a composition of differentiable functions, the total derivatives of the loss can be evaluated by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called (reverse) automatic differentiation (AD).
- AD is not numerical differentiation, nor symbolic differentiation.

Illustration

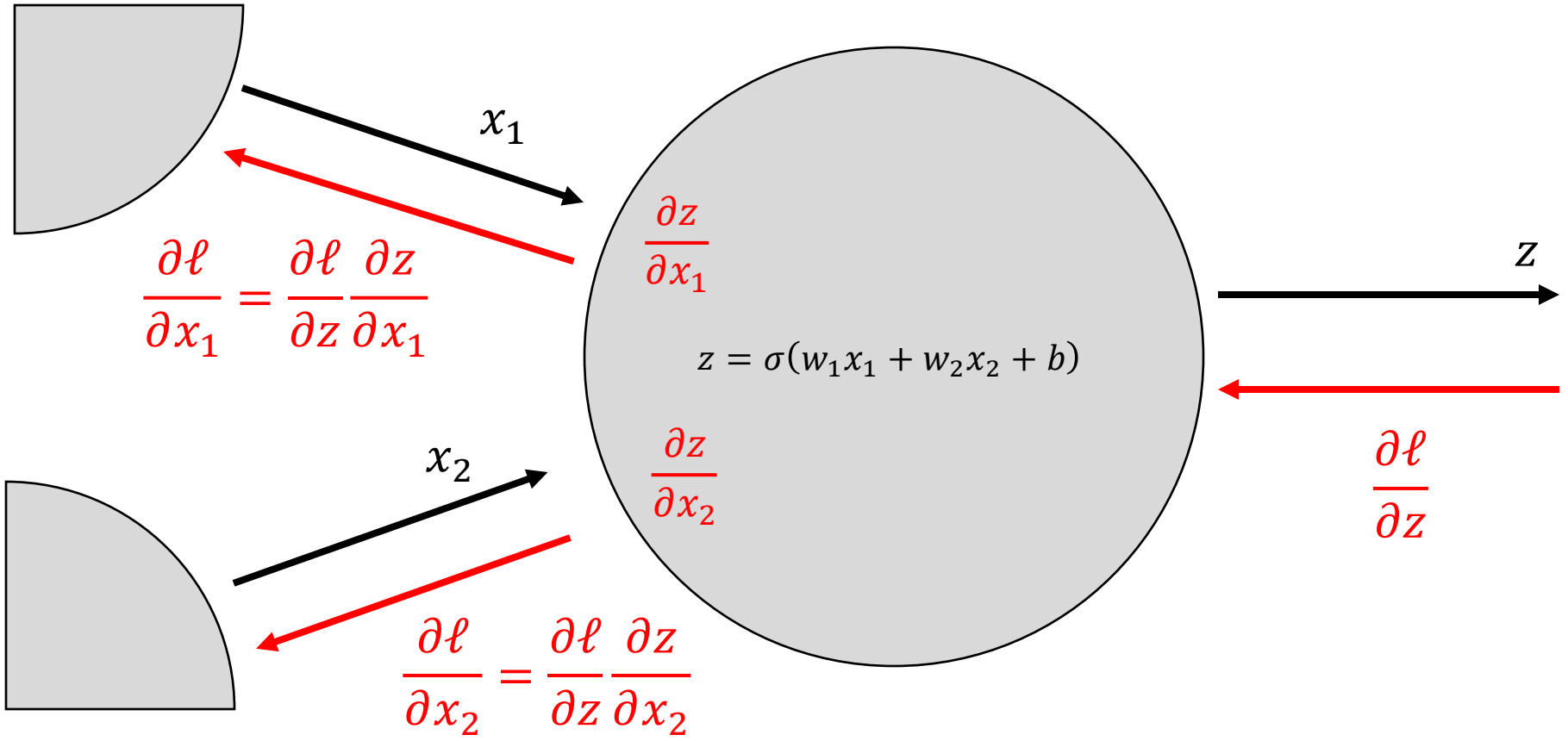
- As an illustration, let us consider a simplified 2-layer MLP and the regularized loss function:

$$\ell(W_1, W_2) = \text{crossentropy}\big(\sigma\big(W_2\sigma(W_1 x)\big), y\big) + \lambda(\|W_1\|_2 + \|W_2\|_2)$$

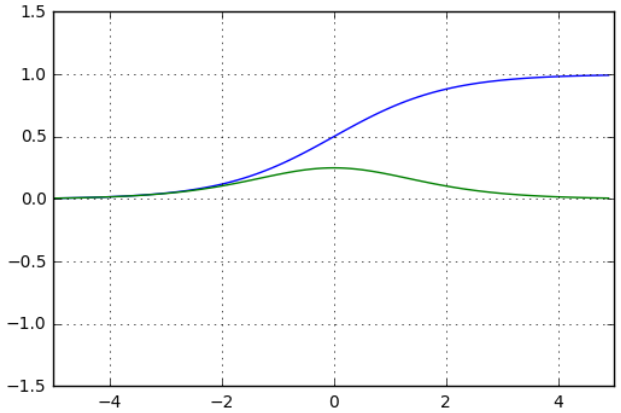


Backpropagation

- Inside a single unit/neuron/function

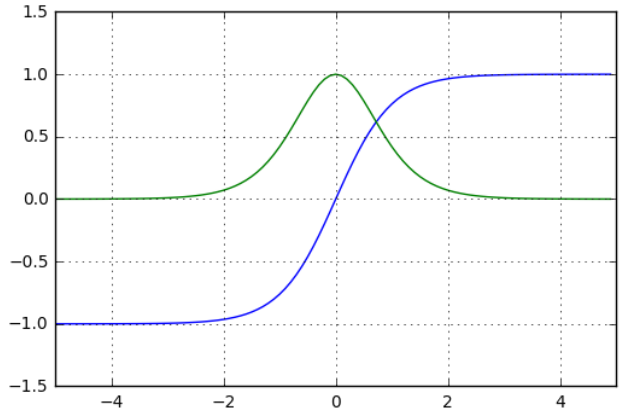


Gradients of activation functions



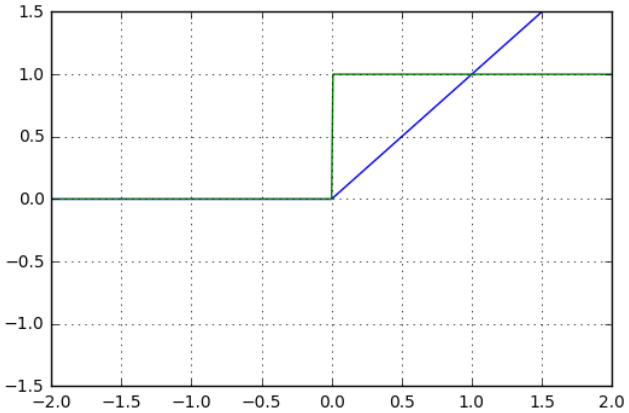
$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x))$$



$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(x) = 1-\tanh(x)^2$$



$$\mathrm{relu}(x) = \max(0,x)$$

$$\mathrm{relu}'(x) = 1_{x>0}$$

- blue: activation function
- green: derivative

Backward pass (Back-propagation)

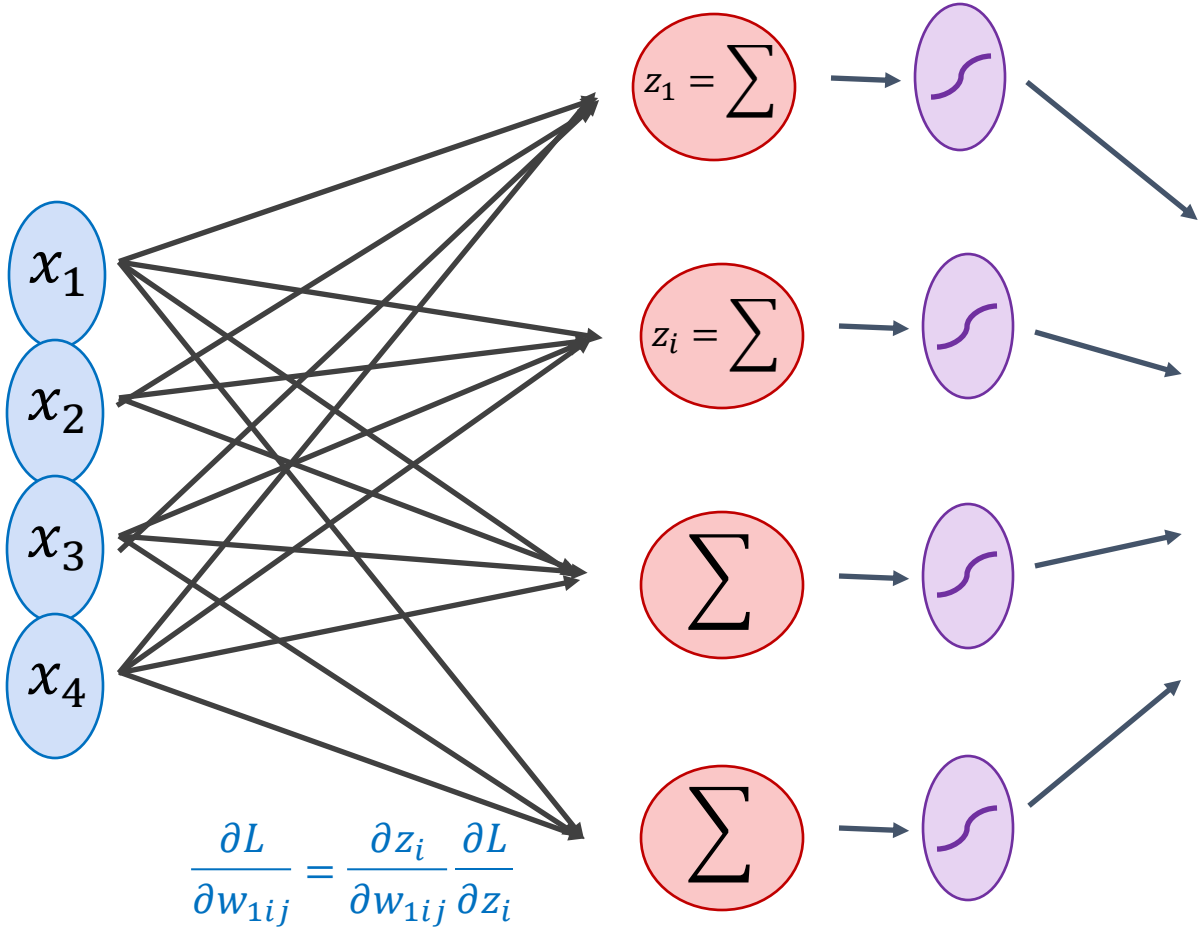
$$\frac{\partial L}{\partial x_k} = \left(\frac{\partial}{\partial x_k}\sum_{j=0}^{n} w_{1ij}x_j + b_1\right)\frac{\partial L}{\partial z_i}$$

$$\frac{\partial L}{\partial z_i} = \frac{\partial\, Sigmoid}{\partial z_i}(z_k)\frac{\partial L}{\partial a_k}$$

GradInputs

$$\frac{\partial L}{\partial a_k} = \left(\frac{\partial}{\partial a_k}\sum_{i=0}^{n} w_{2i}a_i + b_2\right)\frac{\partial L}{\partial p_1}$$

$$\frac{\partial L}{\partial p_1} = \frac{\partial\, Sigmoid}{\partial p_1}(p_1)\frac{\partial L}{\partial \hat{y}_1}(\hat{y}_1, y_1)$$



$$\frac{\partial L}{\partial w_{1ij}} = \frac{\partial z_i}{\partial w_{1ij}}\frac{\partial L}{\partial z_i}$$

$$\frac{\partial L}{\partial w_{2i}} = \frac{\partial p_1}{\partial w_{2i}}\frac{\partial L}{\partial p_1}$$

$$\hat{y}_1 = Sigmoid(p_1) \in [0,1]$$

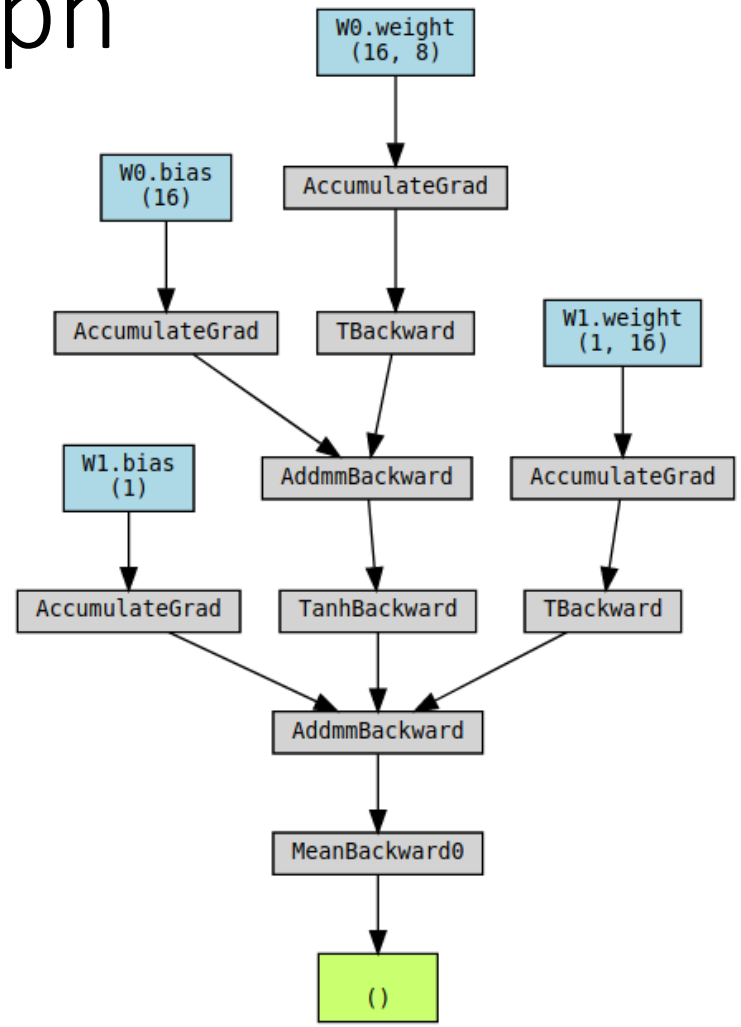
$$\frac{\partial L}{\partial \hat{y}_1} = \frac{\partial}{\partial \hat{y}_1}L(\hat{y}_1, y_1)$$

GradParams

Visualizing the computational graph

- Pytorchviz (package for displaying and debugging computational graph)
- https://github.com/szagoruyko/pytorchviz

```
>>> model = nn.Sequential()
>>> model.add_module('W0', nn.Linear(8, 16))
>>> model.add_module('tanh', nn.Tanh())
>>> model.add_module('W1', nn.Linear(16, 1))
>>> x = torch.randn(1, 8)
>>> y = model(x)
>>> make_dot(y.mean(), params=dict(model.named_parameters()))
```



- Although they are related, the autograd graph is not the network's structure , but the graph of operations to compute the gradient.
- It can be data-dependent and miss or replicate sub-parts of the network.

Autograd in PyTorch

- A Tensor has a Boolean field « requires_grad », set to False by default, which states if PyTorch should build the graph of operations so that gradients w.r.t. to it can be computed.
- The result of a tensorial operation has this flag to True if any of its operand has it to True.

```
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```

- A Tensor also has a field grad, itself a tensor of same size, type, and device (or None) used to accumulate gradients by some functions.

Autograd in PyTorch

- torch.autograd.grad(outputs, inputs) computes and returns the sum of gradients of outputs w.r.t. the specified inputs. This is always a tuple.

- An alternative is torch.autograd.backward(tensors) or Tensor.backward(), which accumulates the gradients in the grad fields of the « leaf » tensors, those which are not results of an operation.

- Using the latter is standard for training models, as it automatically updates gradients for all parameters influencing the loss.

Autograd in PyTorch

- An example: $\ell(x) = \|x\| = \sqrt{x_1^2 + x_2^2 + x_3^2}$, which gives $\frac{\partial \ell}{\partial x_k} = \frac{x_k}{\ell}$
- For $(x_1, x_2, x_3) = (1,2,2\ )$, we have $\ell = 3$ and $\nabla \ell = \left(\frac{1}{3}, \frac{2}{3}, \frac{2}{3}\right)$

- Option 1

```
>>> x = torch.tensor([1., 2., 2.]).requires_grad_()
>>> l = x.norm()
>>> l
tensor(3., grad_fn=<NormBackward0>)
>>> g = torch.autograd.grad(l, (x,))
>>> g
(tensor([ 0.3333, 0.6667, 0.6667]),)
```

- Option 2

```
>>> x = torch.tensor([1., 2., 2.]).requires_grad_()
>>> l = x.norm()
>>> l
tensor(3., grad_fn=<NormBackward0>)
>>> l.backward()
>>> x.grad
tensor([ 0.3333, 0.6667, 0.6667])
```

Autograd in PyTorch

- Autograd can also track the computation of the gradient itself, to allow higher-order derivatives.
- This is specified with create_graph = True:

```
>>> x = torch.Tensor([ 1., 2., 3. ]).requires_grad_()
>>> phi = x.pow(2).sum()
>>> g1 = torch.autograd.grad(phi, x, create_graph = True)
>>> g1
tensor([2., 4., 6.], grad_fn=<ThMulBackward>)
>>> psi = g1[0].exp() - g1[2].exp()
>>> g2 = torch.autograd.grad(psi, x)
>>> g2
tensor([ 14.7781, 0.0000, -806.8576])
```

Only need to write code for the forward pass, backward pass is computed automatically

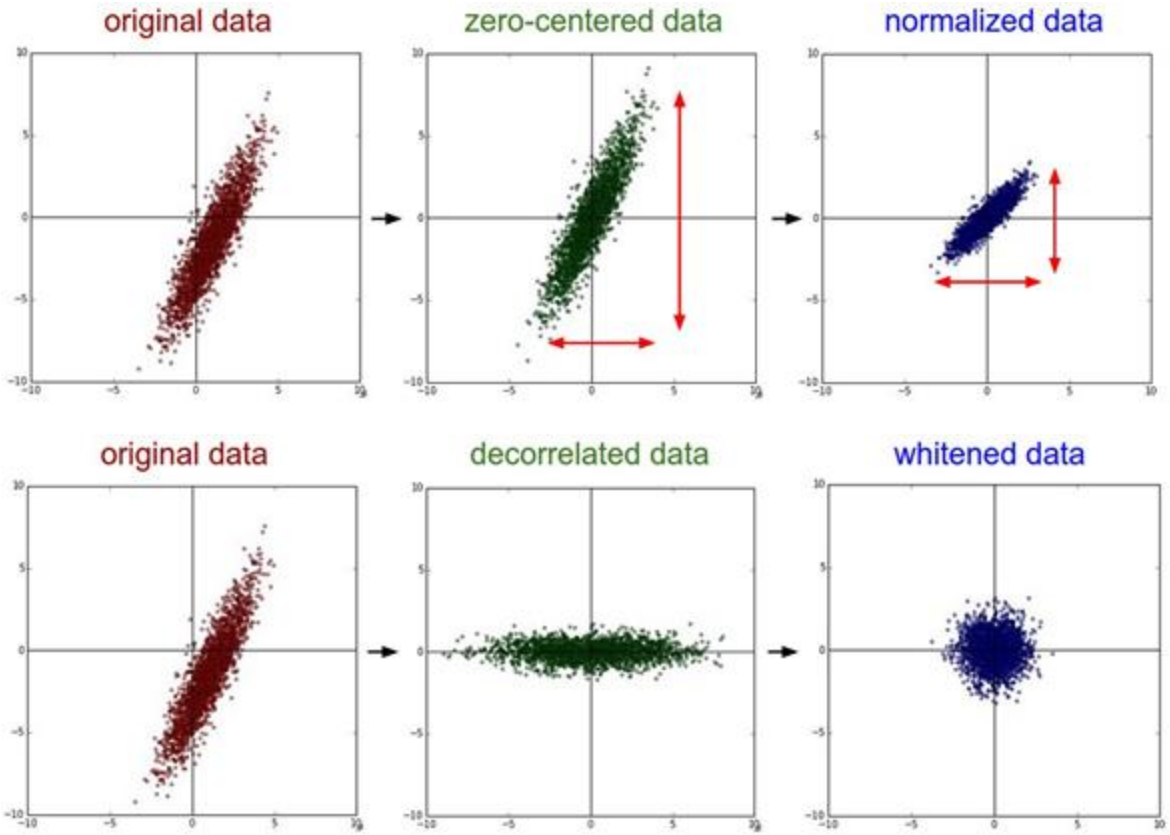
```
# Define a two layer neural network
class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)
```

```
# Construct our loss function and an Optimizer.
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(N):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)
    # Compute
    loss = criterion(y_pred, y)
    # Zero gradients
    optimizer.zero_grad()
    # Perform a backward pass
    loss.backward()
    # Update the weight
    optimizer.step
```

Weight initialization

Data Normalization

- Normalization of the data can be useful



Example: unsupervised learning

Example: supervised learning

Why initialize weights?

- The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network.

- If initial weights are too large, learning the network can take a long time or may fail

- If initial weights are too small, network has difficulty breaking symmetry

- Initial weights should be positive and negative to avoid saturation of activity

Xavier Initialization $w_{ij} \sim \mathcal{N}\left(0, \frac{1}{d}\right)$

- Reasonable initialization for a layer with d neurons: mathematical derivation assumes linear activations but when using the ReLU nonlinearity it breaks

- Assume the output of the layer is

$$y = w_1 x_1 + \cdots + w_d x_d + b$$

- Assuming the x_i's and the w_i's are independent

$$\text{var}(y) = \sum_{k=1}^{d} \text{var}(x_k)\text{var}(w_k) = d\ \text{var}(X)\text{var}(W)$$

by assuming that the x_i's and the w_i's are some i.i.d realizations of random variables X and W

- Hence, if we want that $\text{var}(y) = \text{var}(X)$, we need that $d\ \text{var}(W) = 1$

Xavier Glorot, Yoshua Bengio,Understanding the difficulty of training deep feedforward neural networks, Journal of Machine Learning Research, 2010

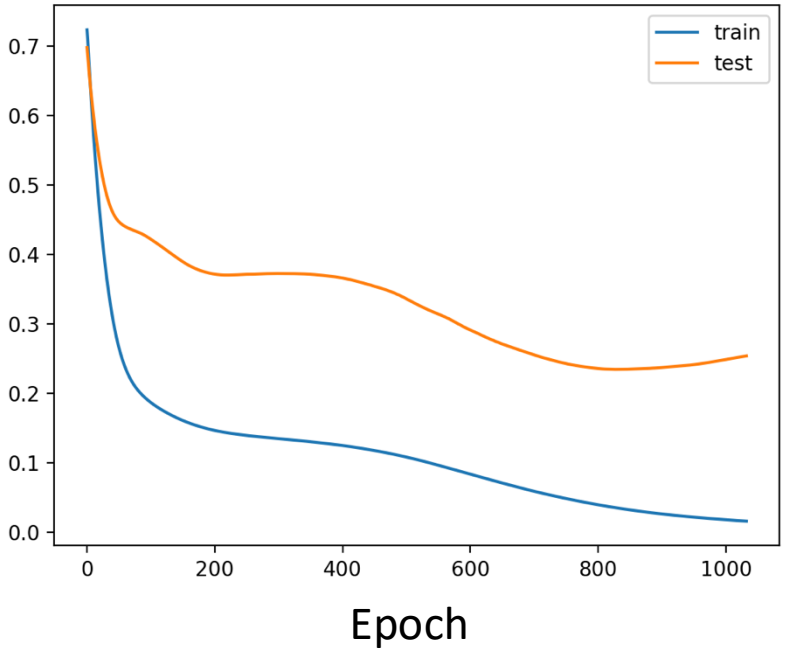
Glorot/Bengio Normalization

- For each layer with n_{in} inputs and n_{out} outputs, the weight from input i to output j should be set as
  - $w_{ij} \sim \text{Gaussian}\left(0, \frac{c^2}{n_{\text{out}}+n_{\text{in}}}\right)$ or $w_{ij} \sim \text{Uniform}\left(-c\sqrt{{}^{6}/_{n_{out}+n_{in}}}, +c\sqrt{{}^{6}/_{n_{out}+n_{in}}}\right)$

- Rationale
  - Xavier scheme controls activation variance
  - Glorot/Bengio aimed to control both activation variance and gradient variance

- Initialization scheme will depend on activation functions you're using
  - Most schemes are focused on logistic, tanh, softmax functions

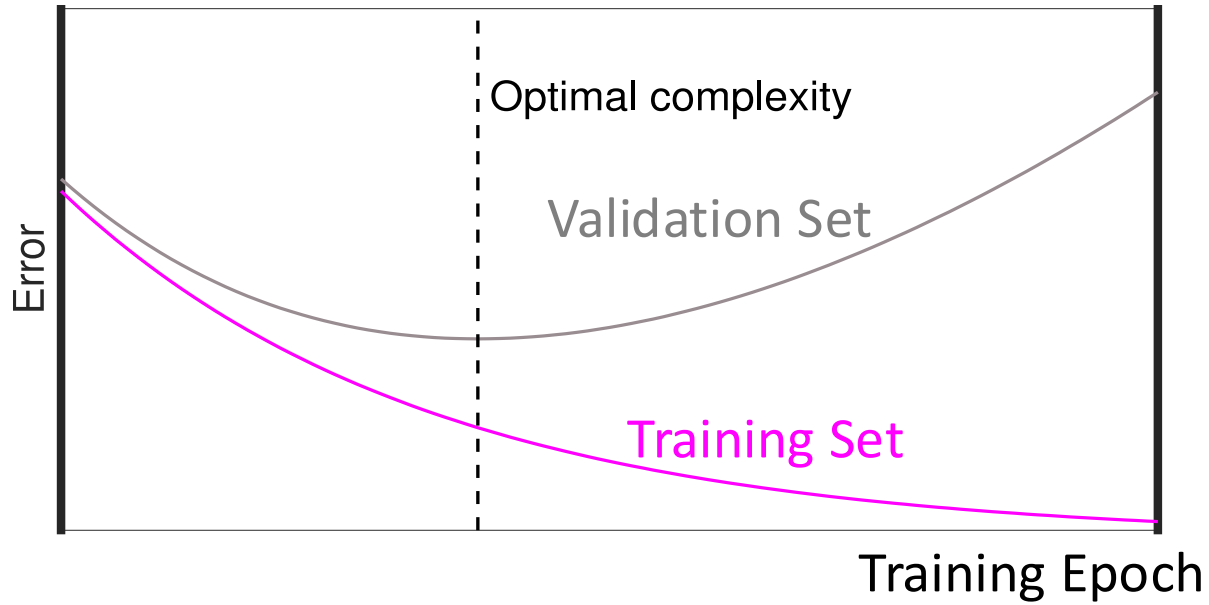
When To Stop Training



- Train n epochs; lower learning rate; train m epochs
  - Bad idea: can't assume one-size-fits-all approach
- Error-change criterion
  - Stop when error isn't dropping "significantly"
  - Bad idea: often plateaus in error even when weights are changing a lot
  - Compromise: criterion based on % drop over a window of, say, 10 epochs
    - 1 epoch is too noisy
    - Absolute error criterion is too problem dependent

When To Stop Training

- Early stopping with a validation set
  - Intuition
    - Hidden units all try to grab the biggest sources of error
    - As training proceeds, they start to differentiate from one another
    - Effective number of free parameters (model complexity) increases with training



Conclusion

Conclusion

- Neural networks are parameterized with a huge number of parameters
- Backpropagation is the usual approach to estimate these parameters
- Automatic differentiation is a key aspect for backpropagation
- Computation graph is a very important concept for the automatic differentiation
- Weight initialization is important
- When to stop the training is not so easy to decide