STAT 5526: Lec 9
Intro to Neural Network

Xin (Shayne) Xing

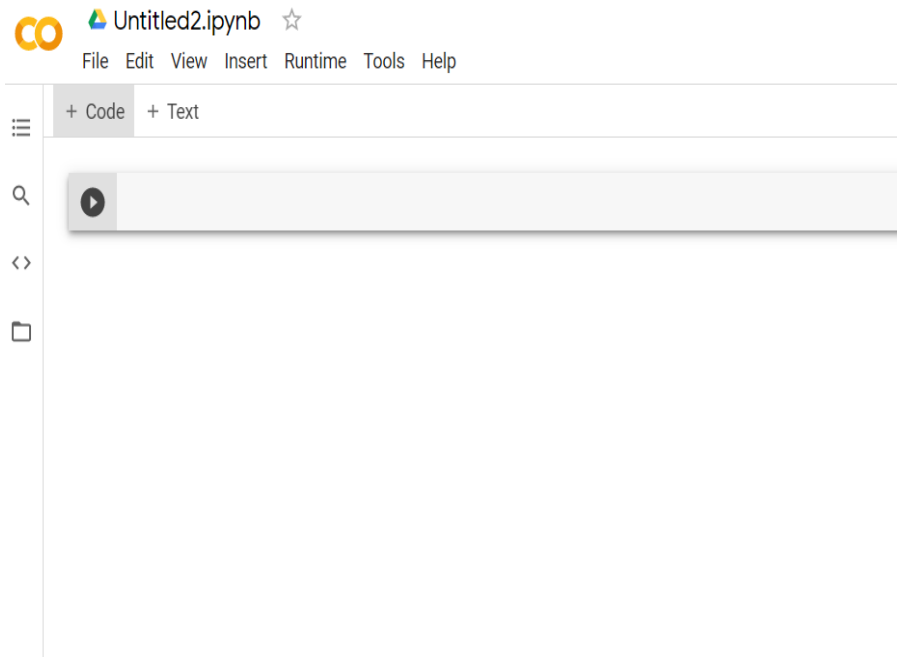
Virginia Tech

Today’s Outlines

1. Setup the Keras
2. Implementation

Setup keras in Google Colab

- A Jupyter notebook lets you write and execute Python (or R) code *locally* in your web browser. Jupyter notebooks make it very easy to tinker with code and execute it in bits and pieces; for this reason they are widely used in scientific computing.
- Enter https://colab.research.google.com/#create=true&language=rin your browser.

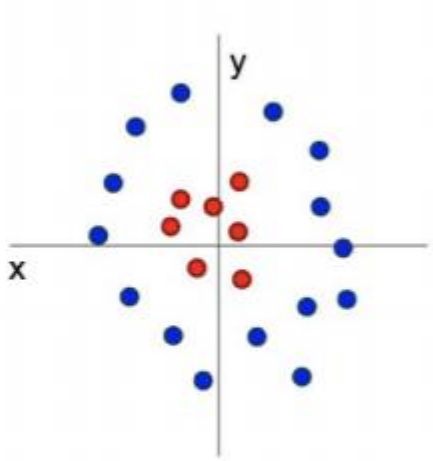


Limitation of Linear Classifiers

Visual Viewpoint

Linear classifiers learn one template per class

Geometric Viewpoint



Linear classifiers can only draw linear decision boundaries

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

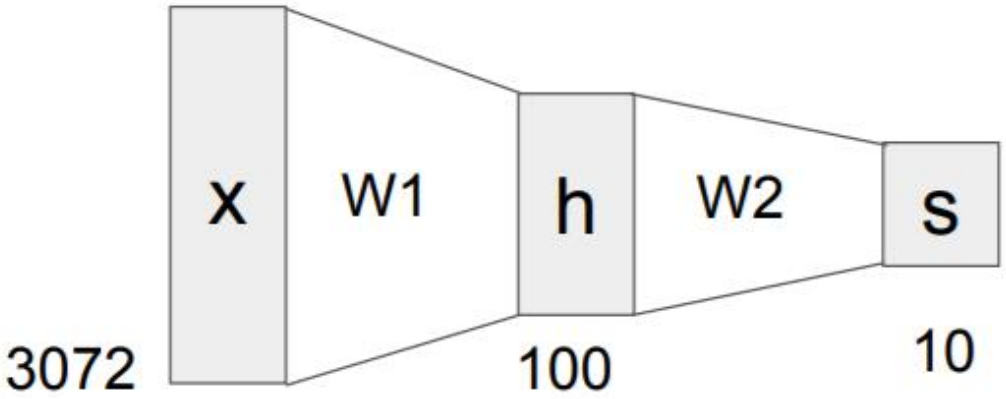
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

"Neural Network" is a very broad term; these are more accurately called "fully-connected networks" or sometimes "multi-layer perceptrons" (MLP)

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



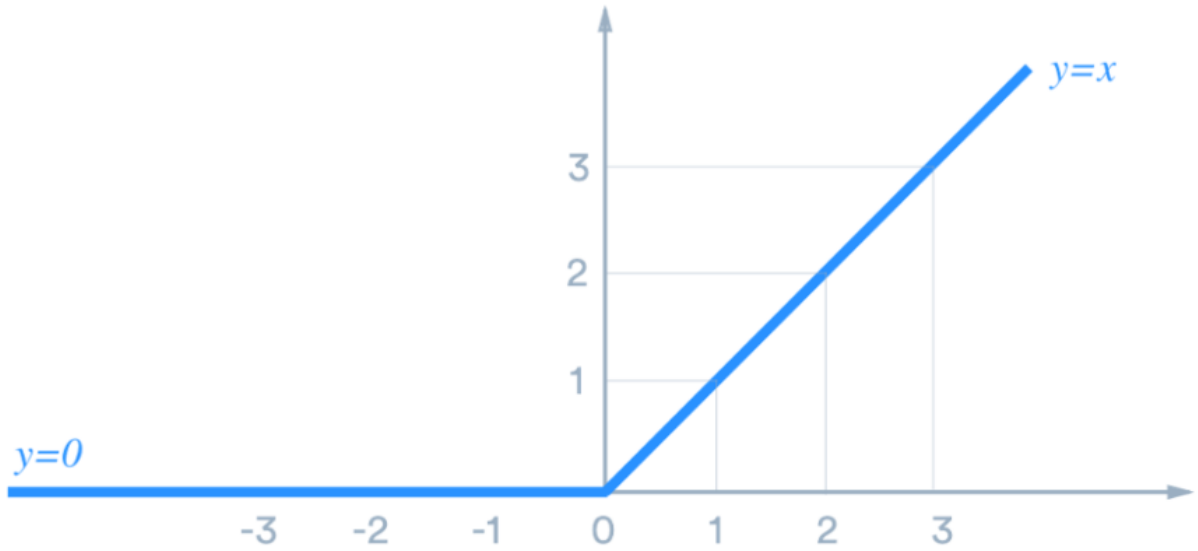
$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$

Activation function

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

- The function is called the **Rectified Linear Unit (ReLU)** activation function.



Activation function

- The function is called the activation function.

- Q: What if we try to build a neural network without one?

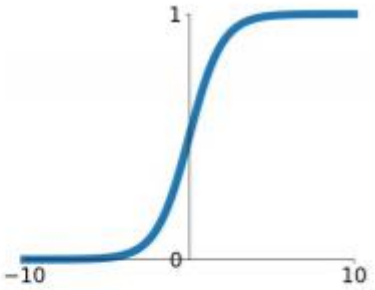
$$f = W_2 W_1 x \qquad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

- We end up with a linear classifier again!

Activation Functions

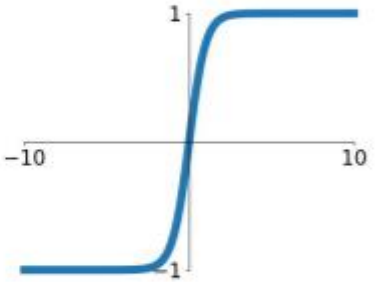
Sigmoid

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



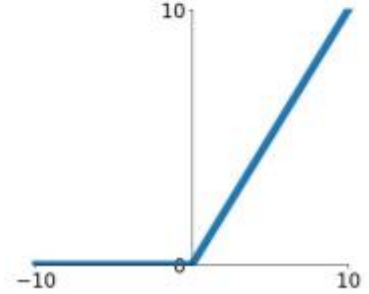
tanh

$$\tanh(x)$$



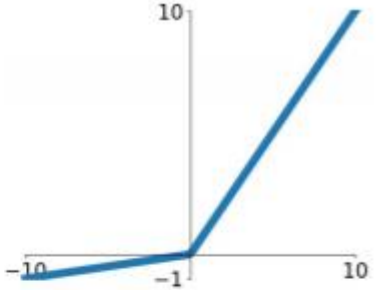
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

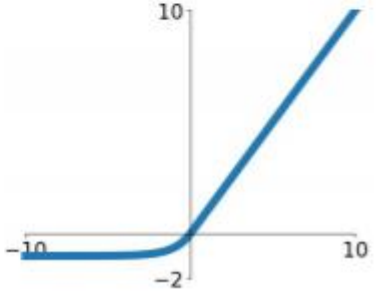


Maxout

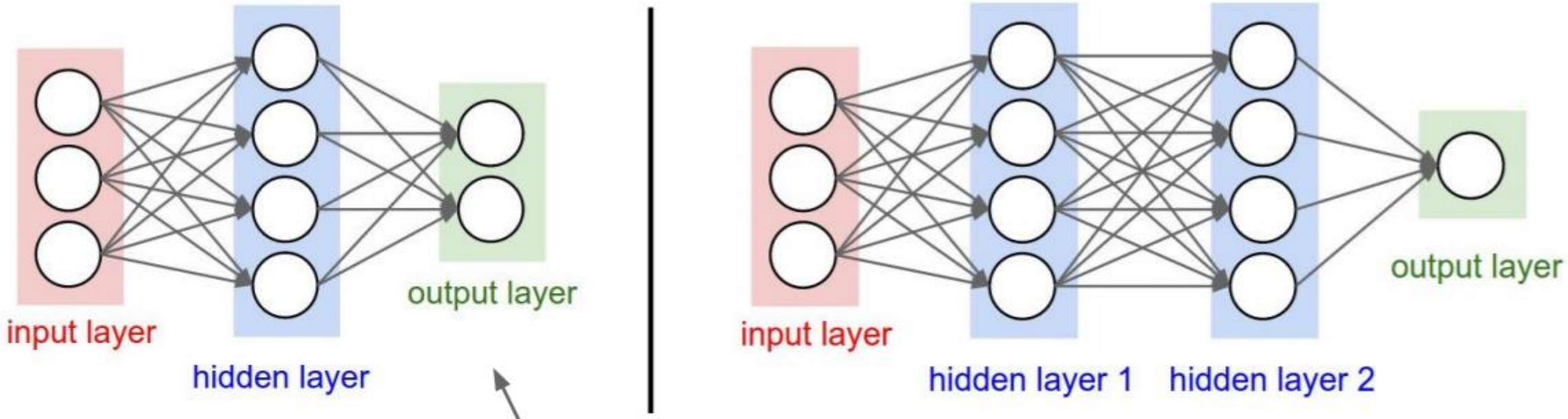
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Architectures



"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

Full implementation of training a 2-layer Neural Network needs ~20 lines:

- 1. Preparing the Data

```
library(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

# reshape
dim(x_train) <- c(nrow(x_train), 784)
dim(x_test) <- c(nrow(x_test), 784)
# rescale
x_train <- x_train / 255
x_test <- x_test / 255


y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

Implementation

- 2. Defining the Model

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = "relu", input_shape = c(784)) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```

Implementation

summary(model)

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 256)               200960
 dense_1 (Dense)             (None, 128)               32896
 dense (Dense)               (None, 10)                1290
=================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
```

Implementation

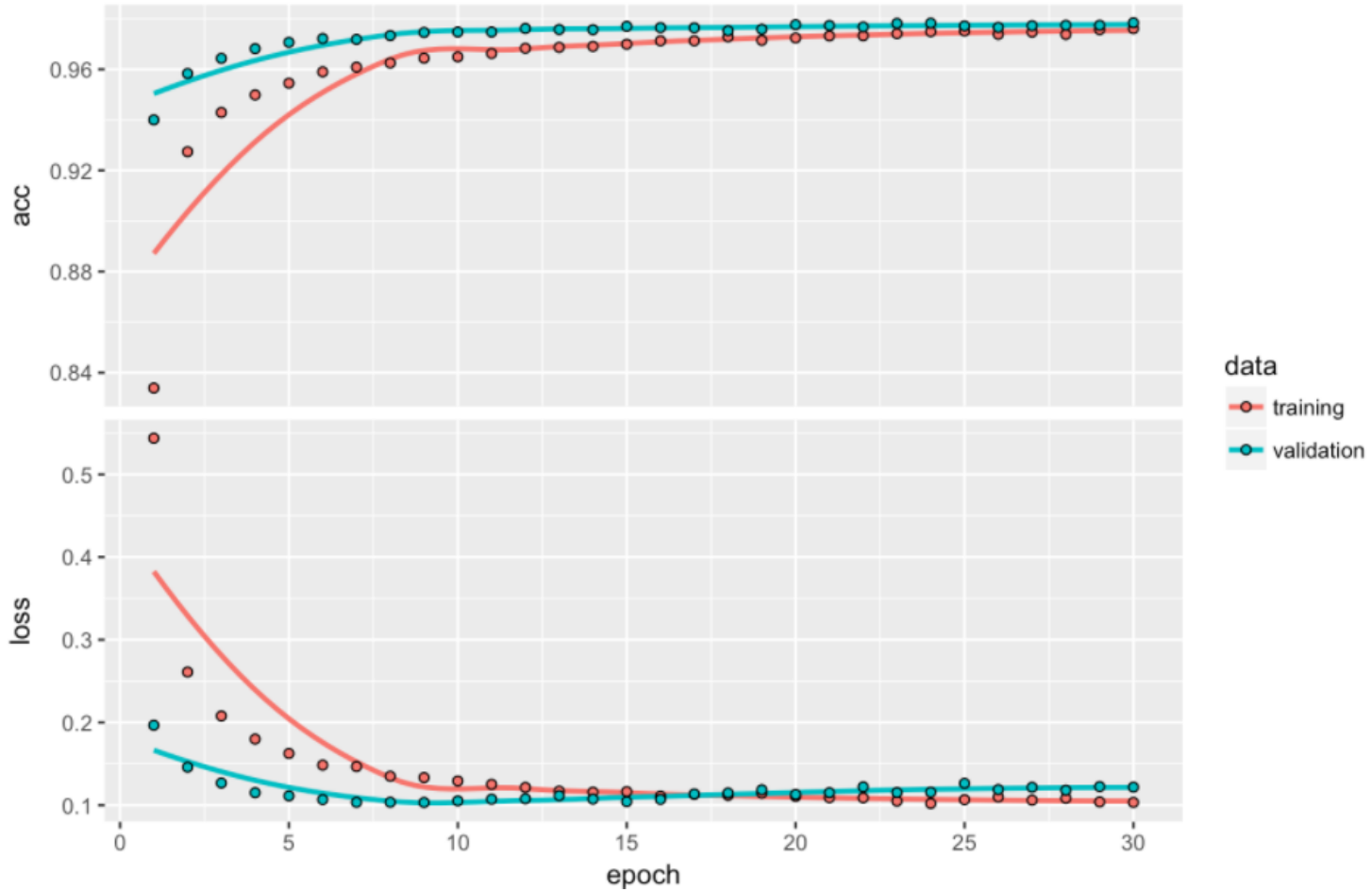
- 3. Set the Optimizer and Train MLP

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(),
  metrics = c("accuracy")
)


history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

Implementation

```
plot(history)
```



Implementation

- 4. Evaluate and Predict on Test Data

```
model %>% evaluate(x_test, y_test,verbose = 0)

 $loss
 [1] 0.1149

 $acc
 [1] 0.9807
```

```
model %>% predict_classes(x_test)

  [1] 7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7 1 2
 [40] 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9 1 7 3 2
 [79] 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9
 [ reached getOption("max.print") -- omitted 9900 entries ]
```

Confusion matrix for calculating accuracy

```
library(caret)
expected_value <- factor(c(1,0,1,0,1,1,1,0,0,1))
predicted_value <- factor(c(1,0,0,1,1,1,0,0,0,1))
example <- confusionMatrix(data=predicted_value, reference =
expected_value)
```

```
Confusion Matrix and Statistics

          Reference
Prediction 0 1
         0 3 2
         1 1 4

               Accuracy : 0.7
```

Implementation

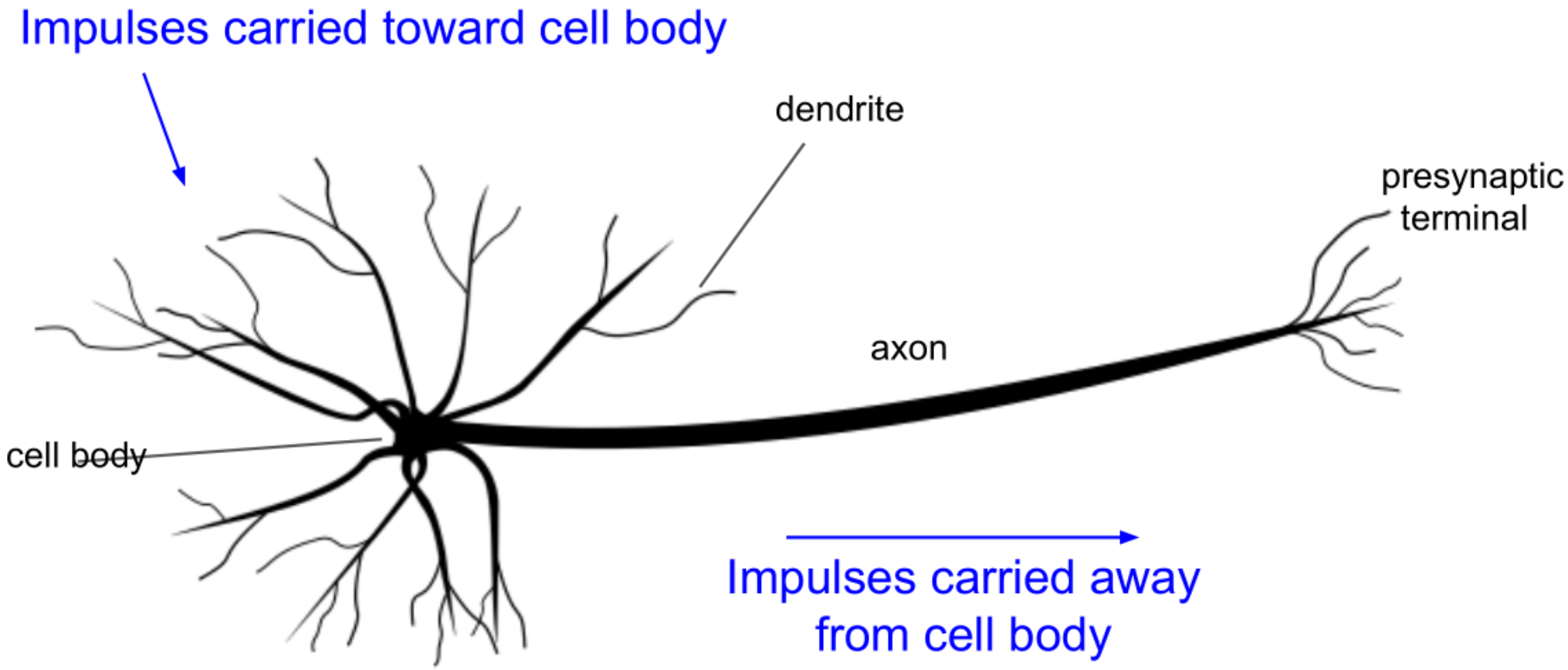
- Summary

1. Preparing the Data
2. Defining the Model
3. Set the Optimizer and Train MLP
4. Evaluate and Predict on Test Data

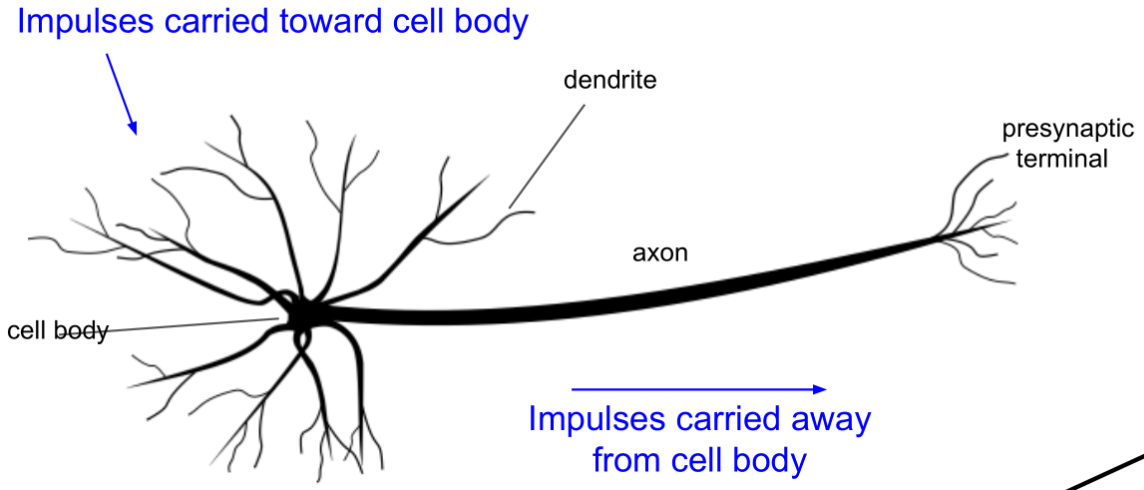
Today’s Outlines

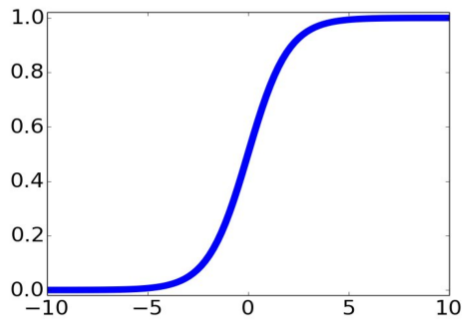
1. Biological Neurons
2. Backpropagation

Neuron



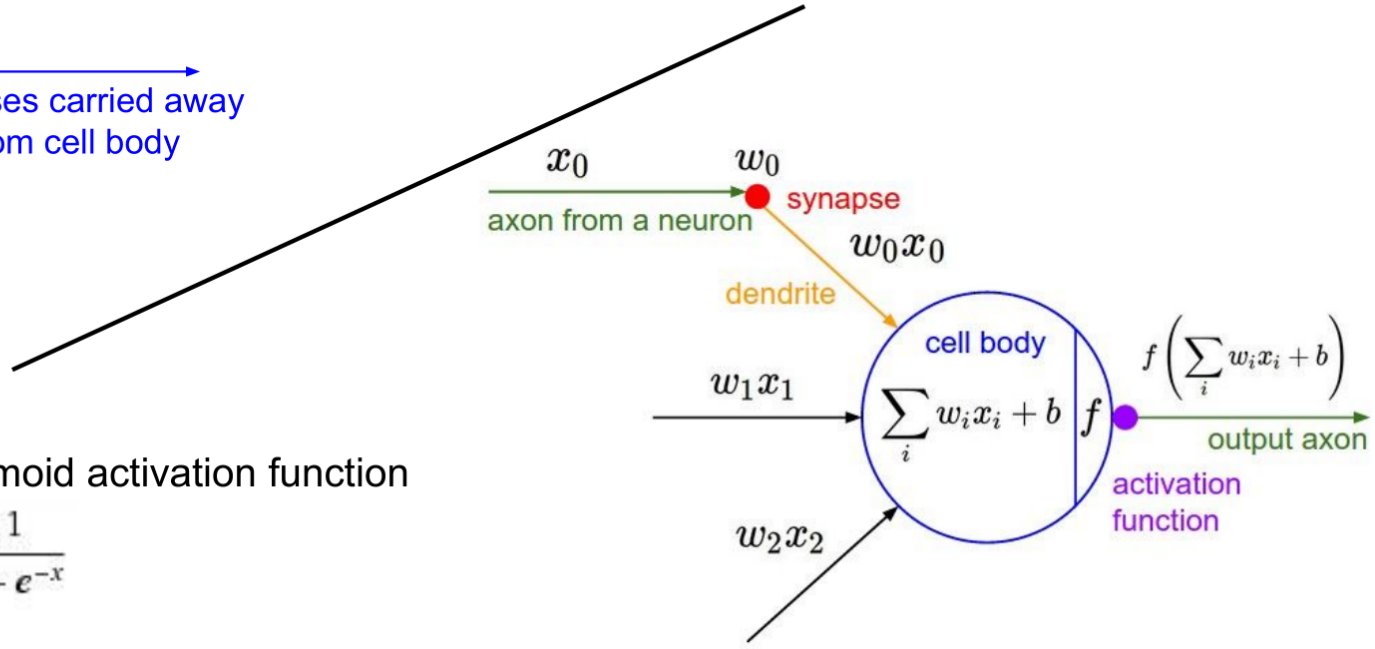
Biological Neuron vs. Artificial Neuron





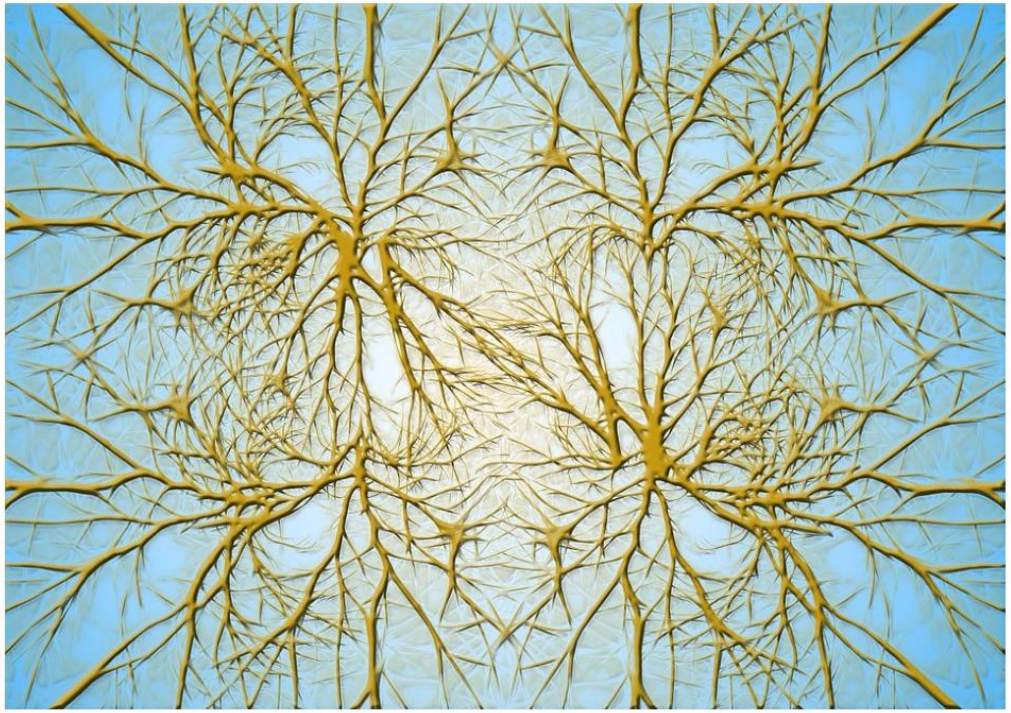
sigmoid activation function

$$\frac{1}{1+e^{-x}}$$

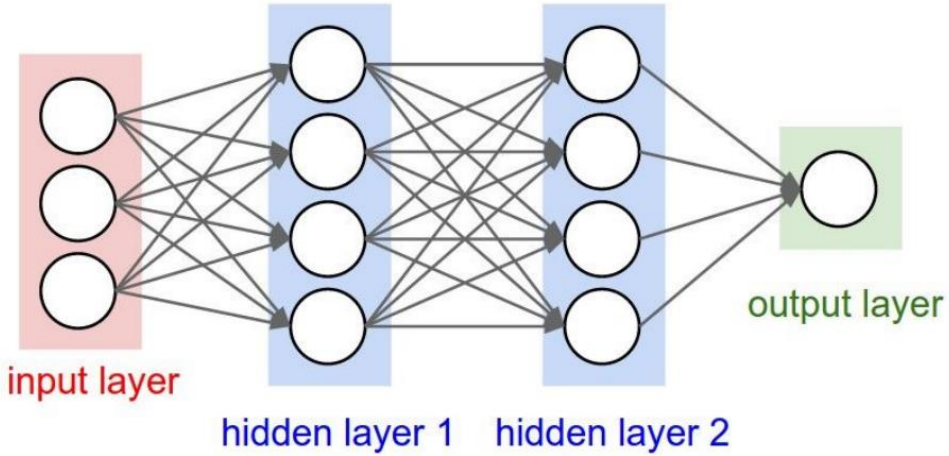


Biological Connectivity

Biological Neurons:
Complex connectivity patterns



Neurons in a neural network:
Organized into regular layers for computational efficiency

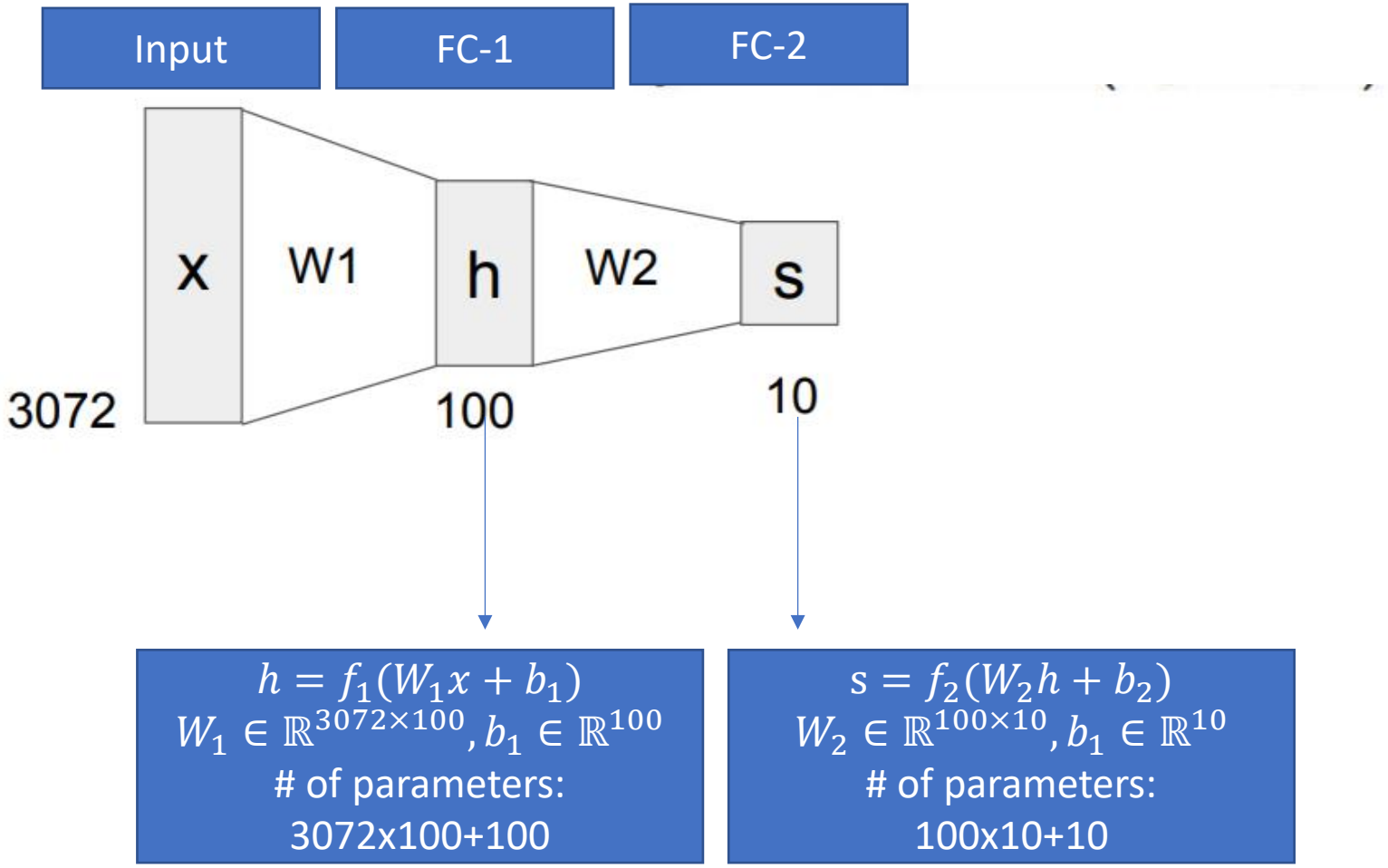


Be very careful with your brain analogies!

Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

Number of parameters



Loss Function

- A loss function tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and
y_i is (integer) label

Loss over the dataset is a
average of loss over examples:

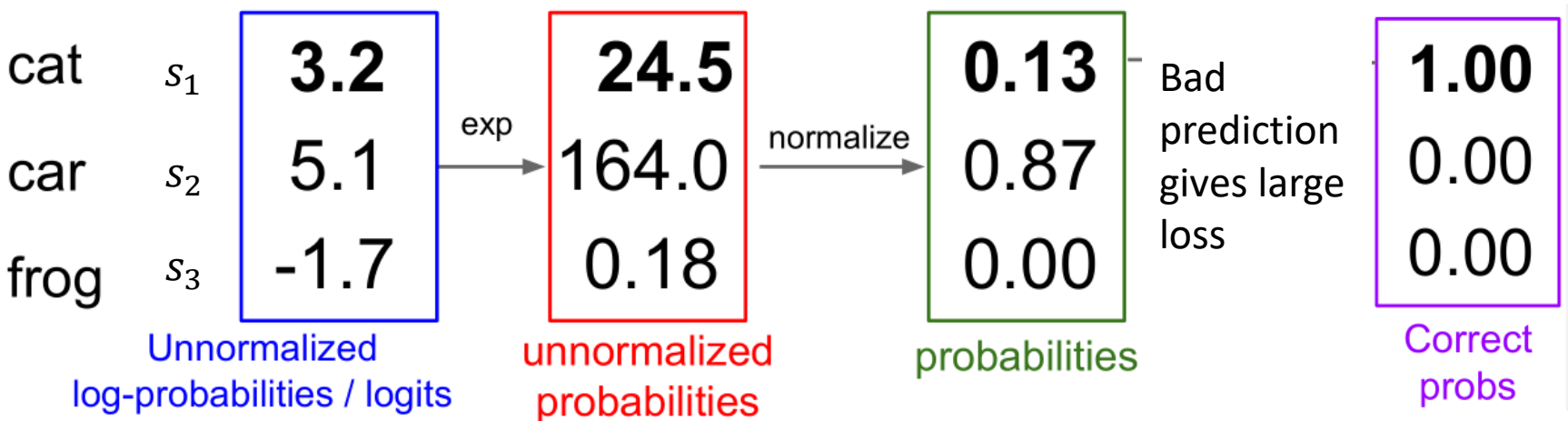
$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Loss function in classification

$$L_i = -\log P(Y = y_i | X = x_i)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function

$$s_k = x^T \beta_k, s_{y_i} = x^T \beta_{y_i}$$



Computing the Gradient

- We can compute the *best* direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descend (at least in the limit as the step size goes towards zero).

- This direction will be related to the **gradient** of the loss function.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Computing the Gradient

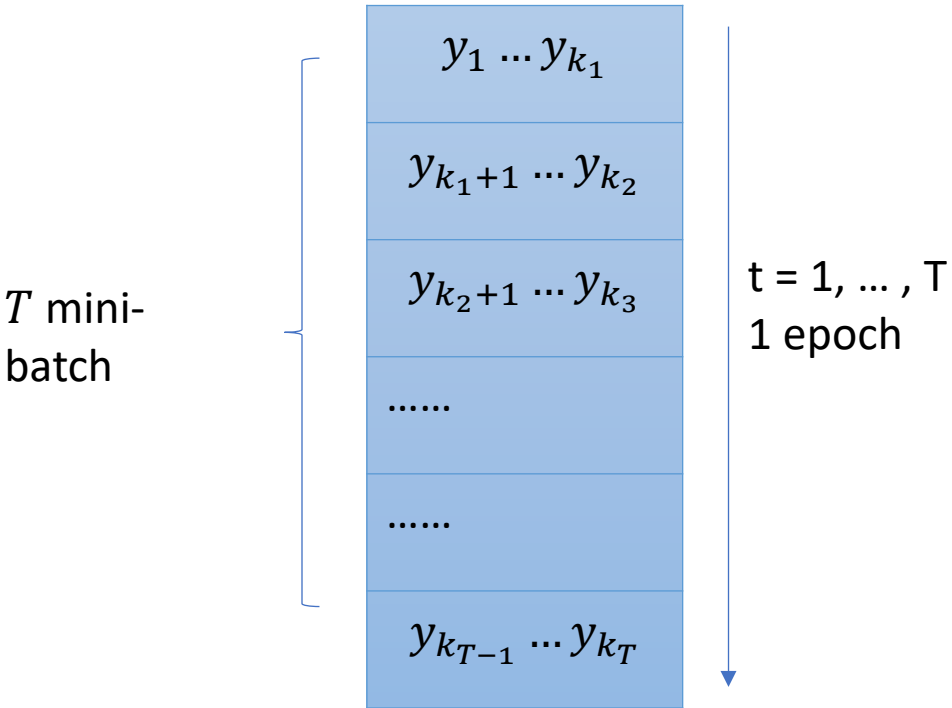
$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

Full sum expensive when N is large!

Stochastic Gradient Decent with Mini-batch

- In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update



$$L_t(W) = \frac{1}{k_t - k_{t-1}} \sum_{i=k_{t-1}+1}^{k_t} L_i(x_i, y_i, W)$$

$$\nabla L_t(W) = \frac{1}{k_t - k_{t-1}} \sum_{i=k_{t-1}+1}^{k_t} \nabla_W L_i(x_i, y_i, W)$$

$$W_{t+1} = W_t + \eta \nabla L_t(W_t)$$

Step Size

- **Effect of step size**. The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step.

- Choosing the step size (also called the *learning rate*) will become one of the most important hyperparameter settings in training a neural network.

$$W_{t+1} = W_t + \eta \nabla L(W_t)$$

where $\nabla L(W) = \left(\frac{\partial L}{\partial w_1}, \cdots, \frac{\partial L}{\partial w_p} \right)$ and p is the total number of parameters.

Backpropagation

- a simple example

$$f(x, y, z) = (x + y)z$$

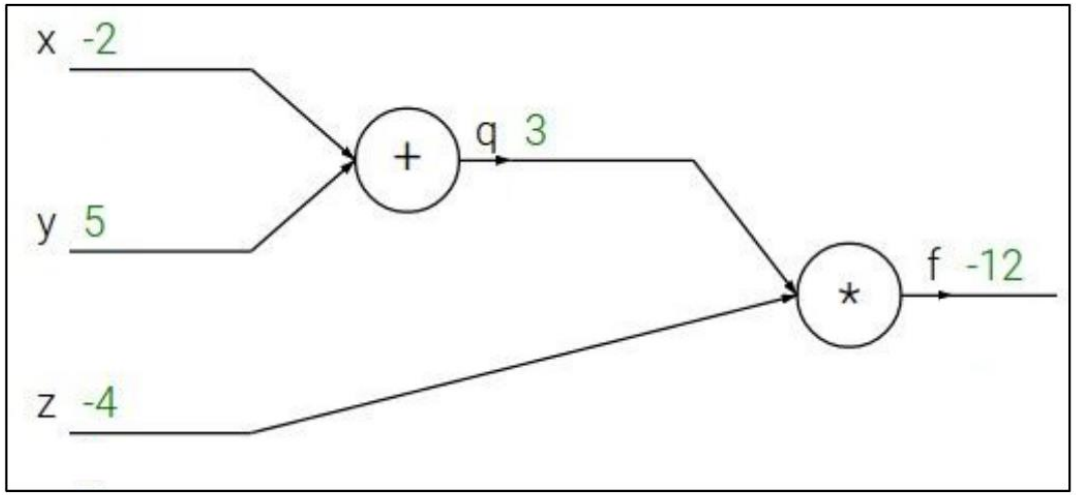
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Graph Representation

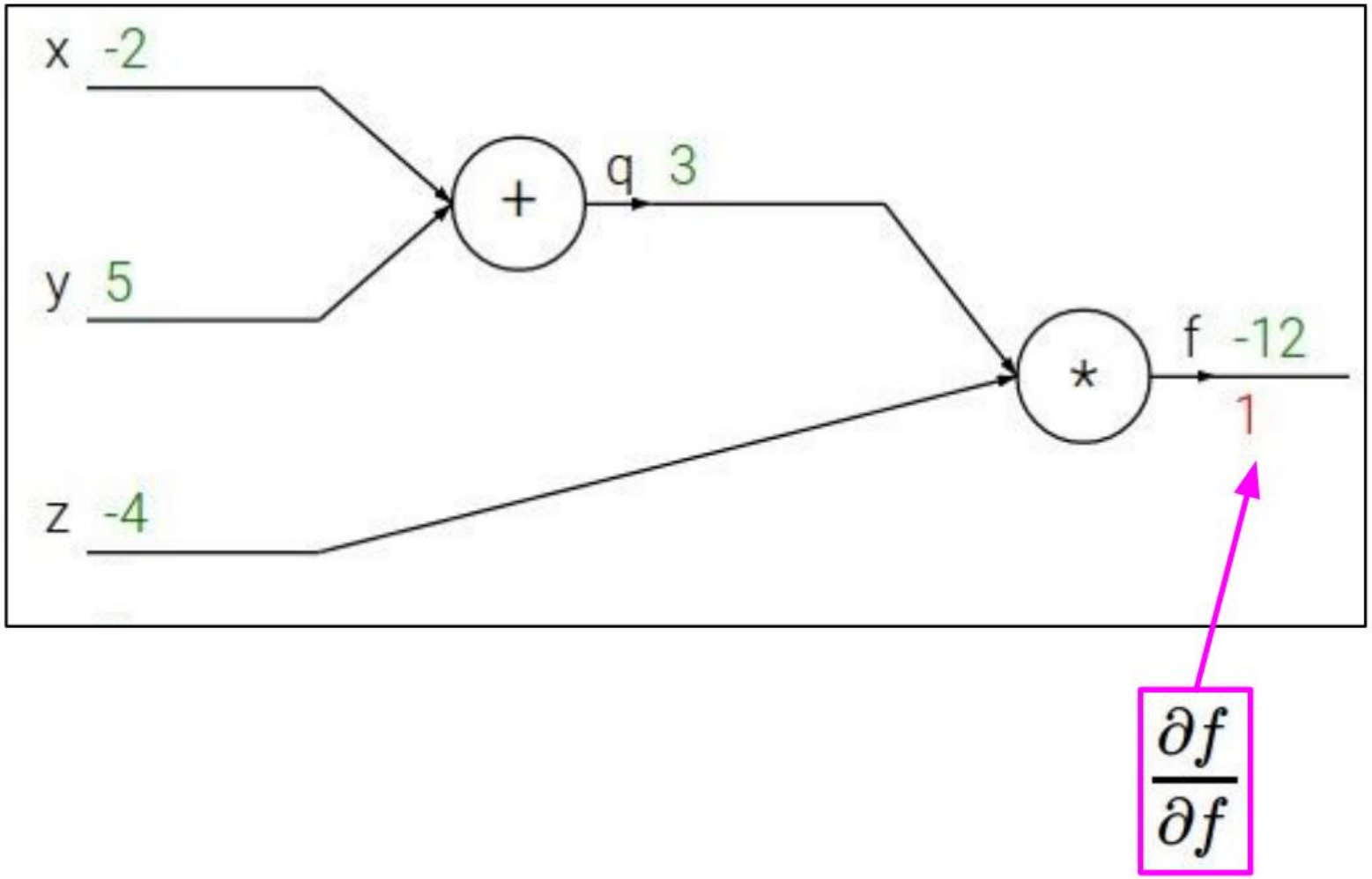
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

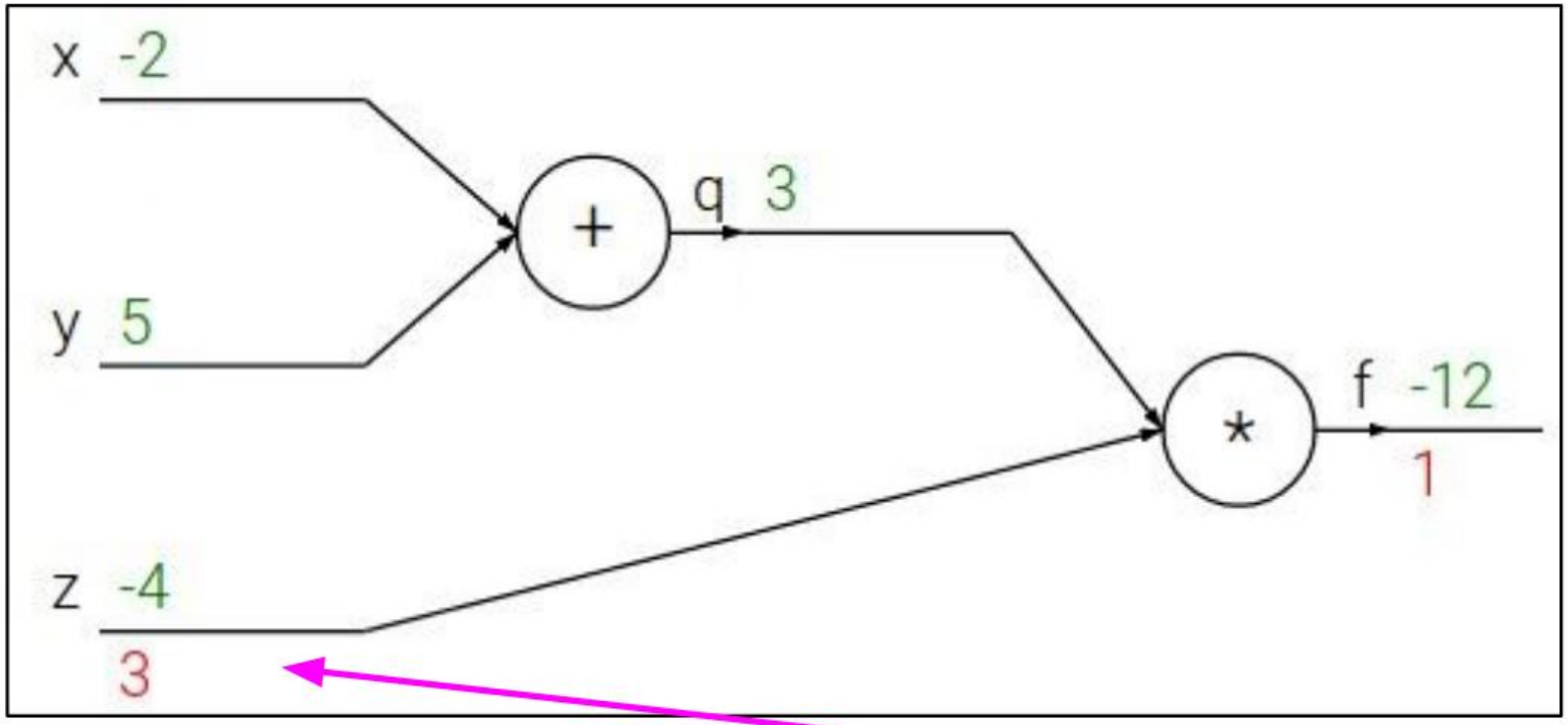
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Top-Down Calculation 1

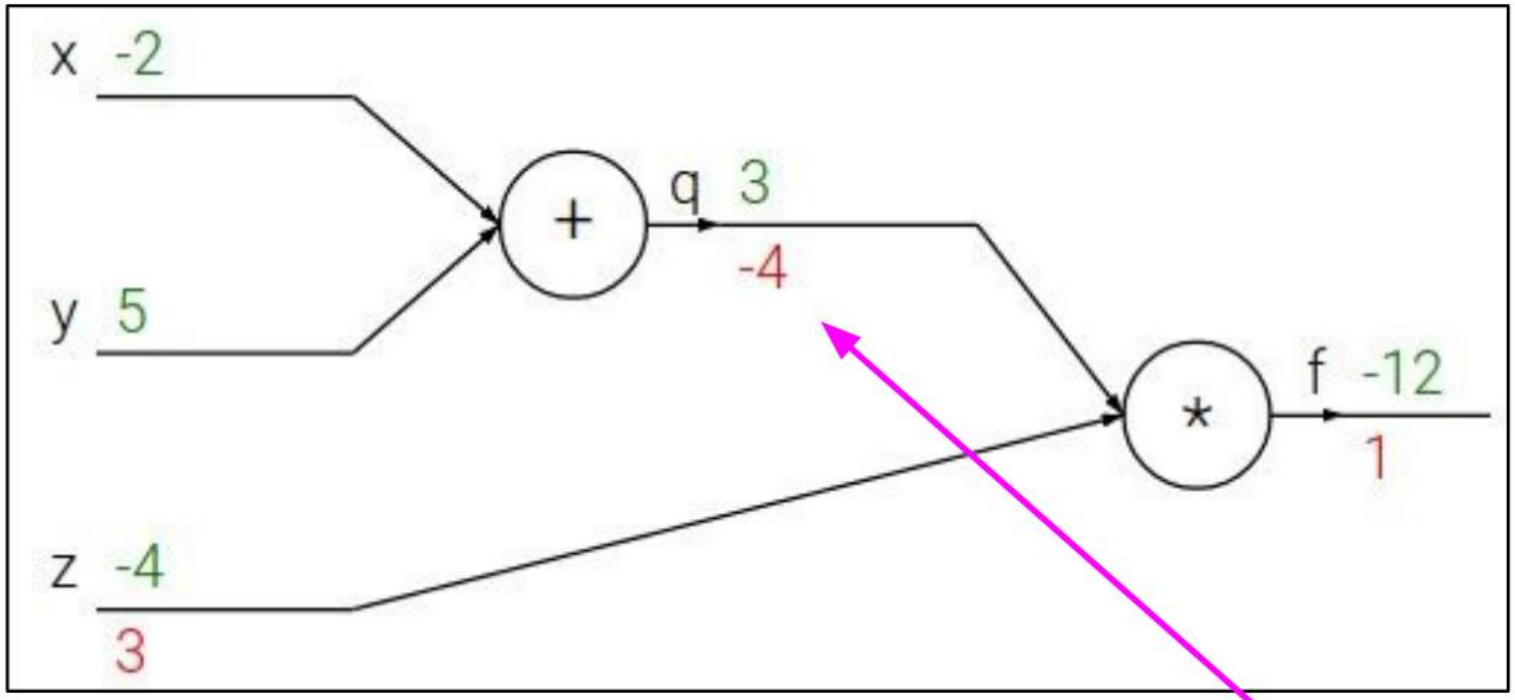


Top-Down Calculation 2



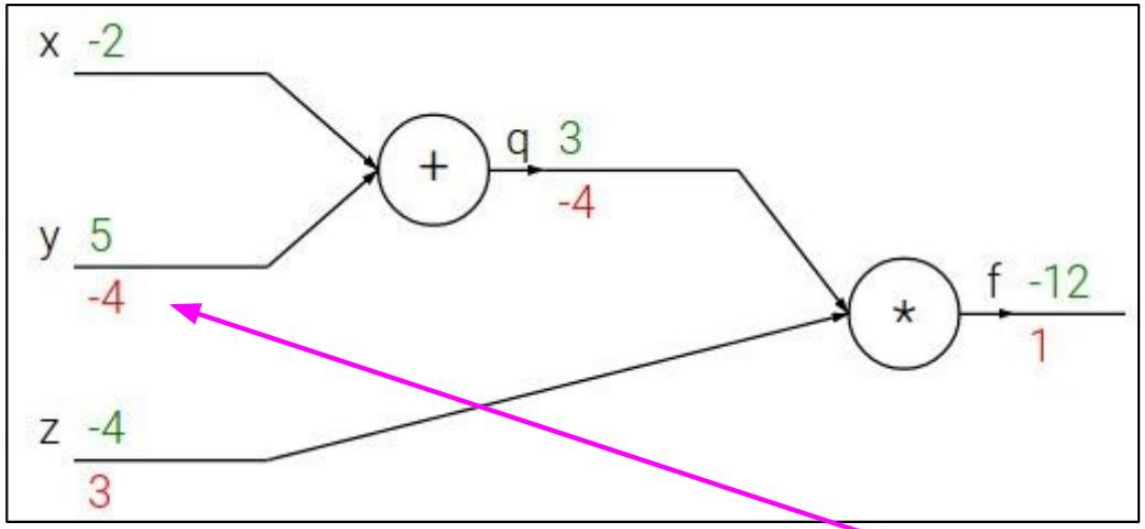
$$\frac{\partial f}{\partial z} = q$$

Top-Down Calculation 3



$$\frac{\partial f}{\partial q} = z$$

Top-Down Calculation 4



$$\frac{\partial f}{\partial y}$$

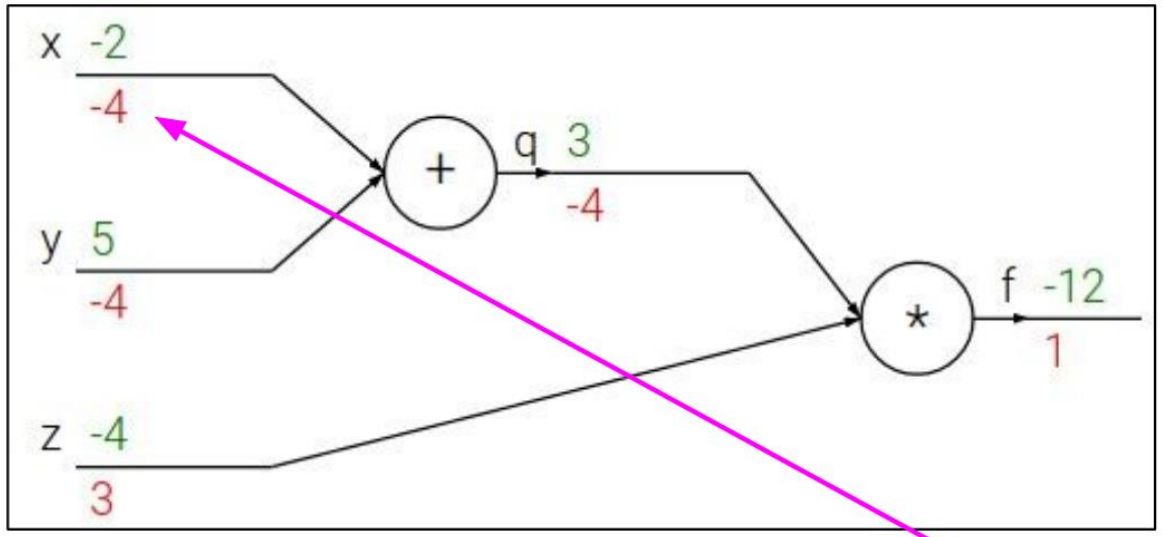
Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\,\frac{\partial q}{\partial y}$$

Upstream gradient

Local gradient

Top-Down Calculation 5



$$\frac{\partial f}{\partial x}$$

Chain rule:

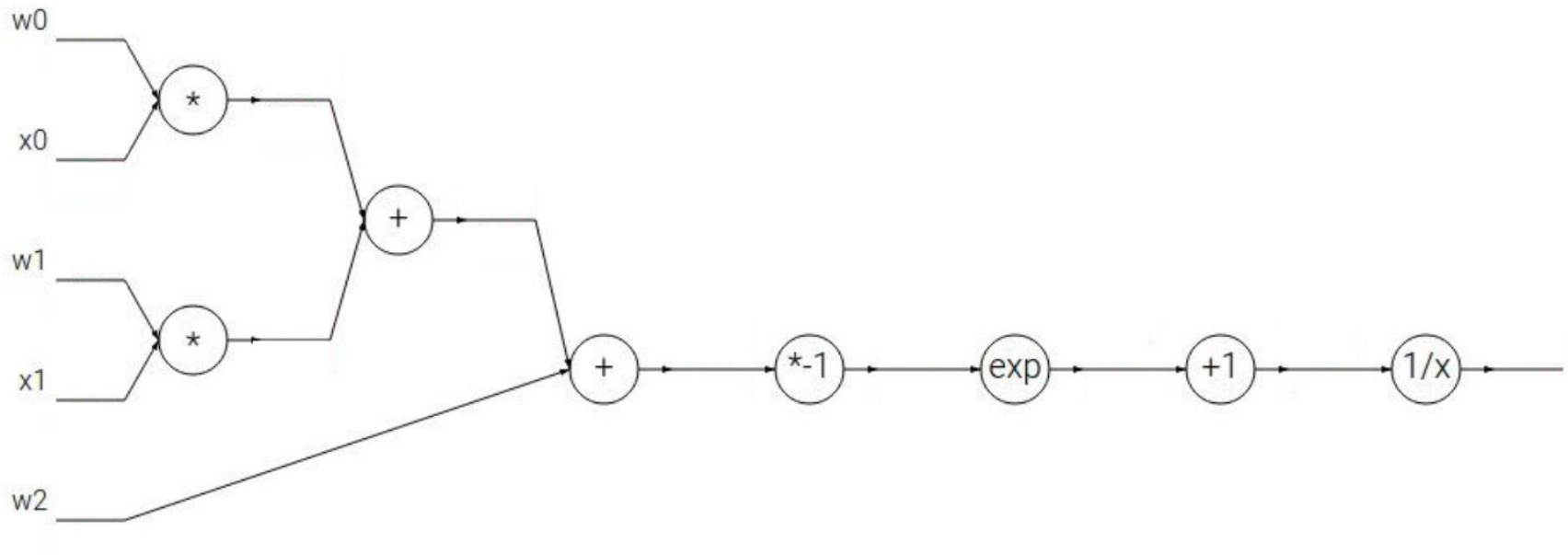
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Upstream gradient

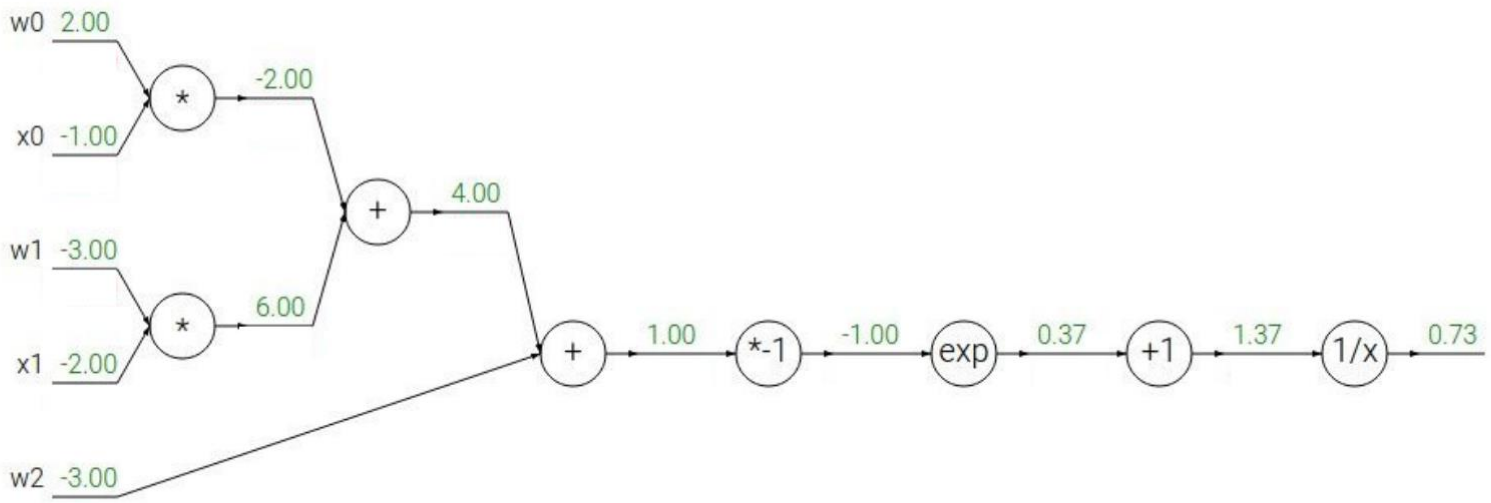
Local gradient

Another Example

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Graph representation



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

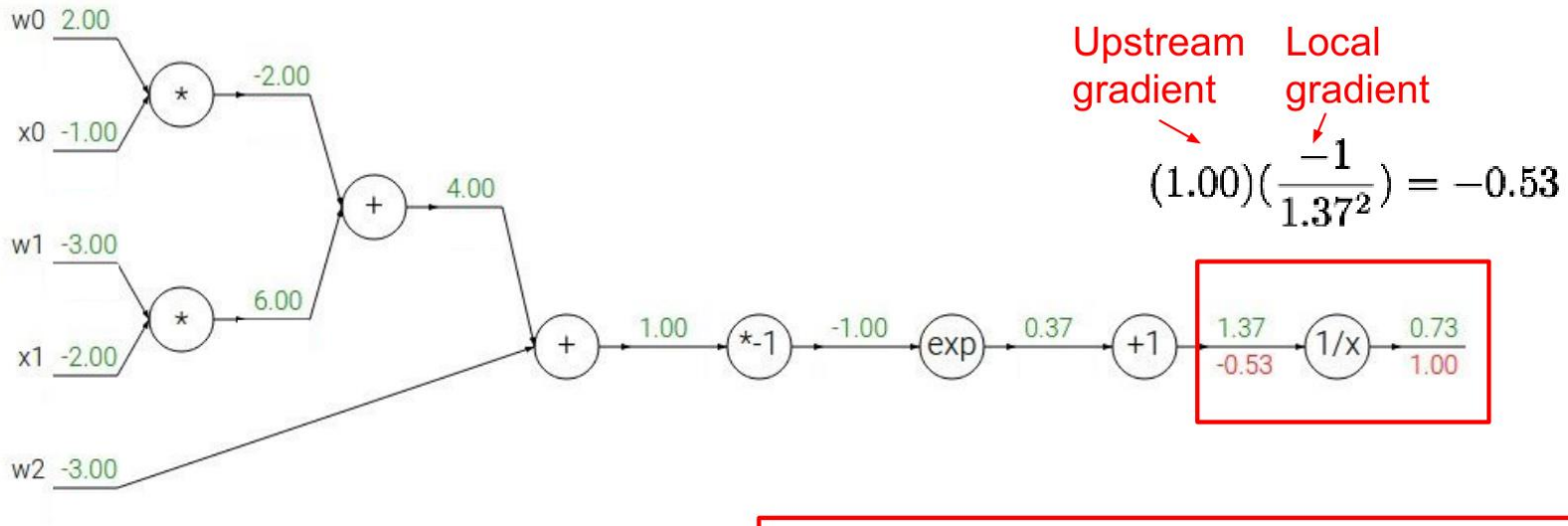
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Top-Down Calculation 1

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

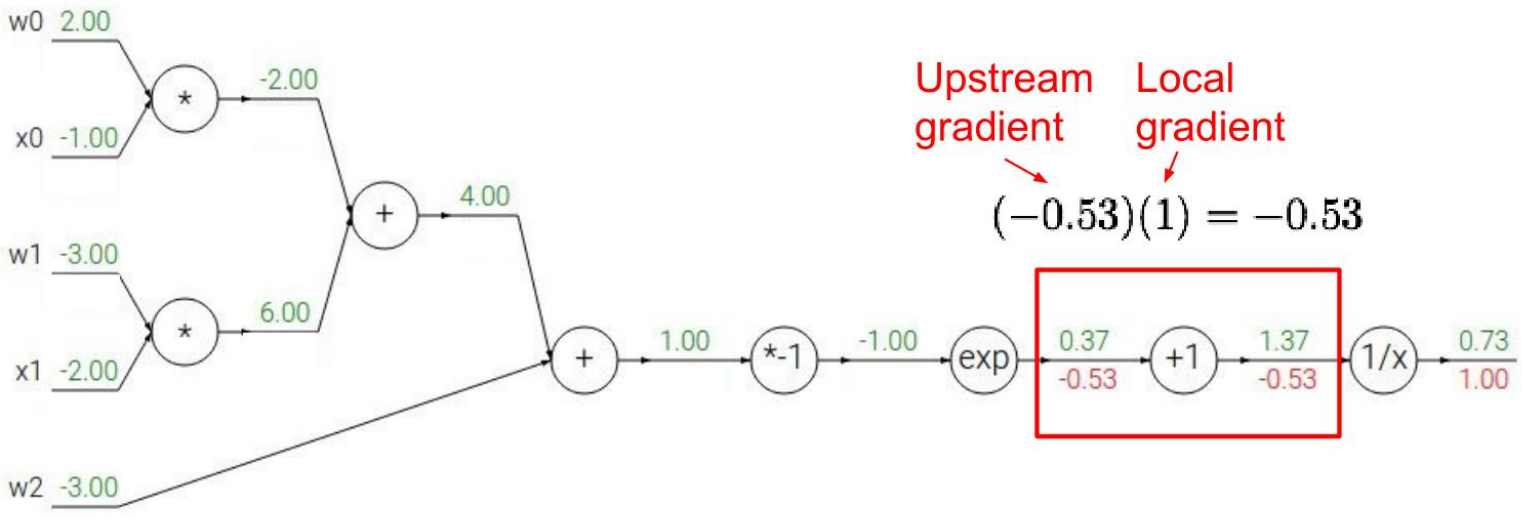
$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

Top-Down Calculation 2

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



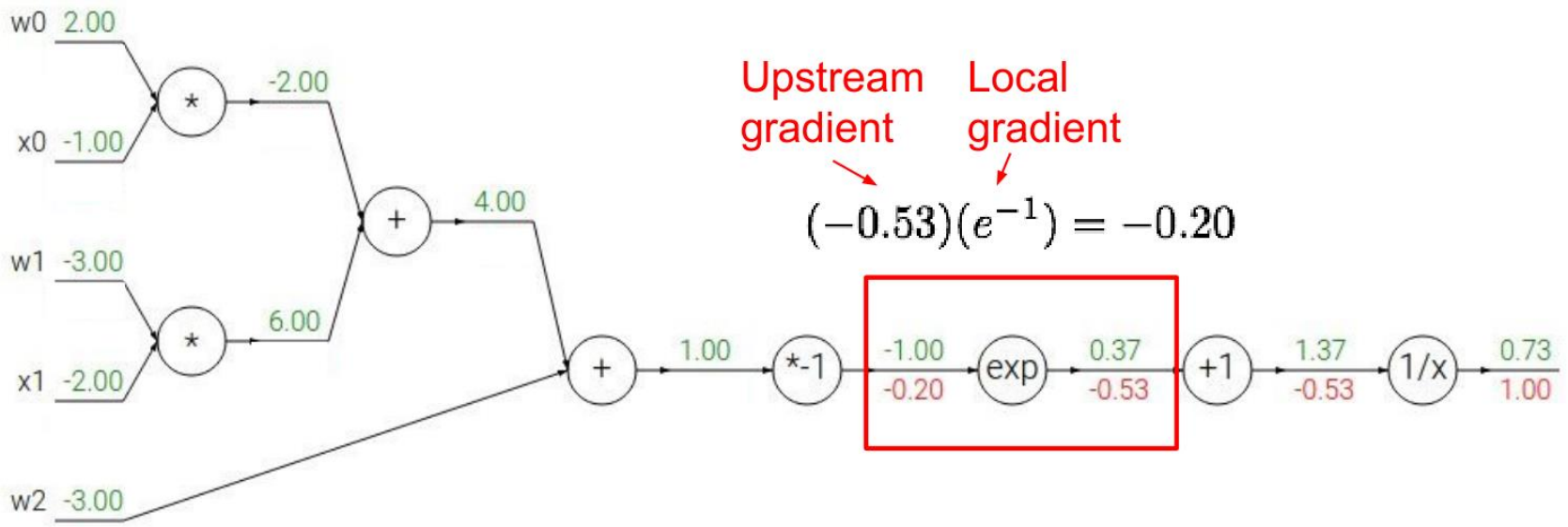
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Top-Down Calculation 3



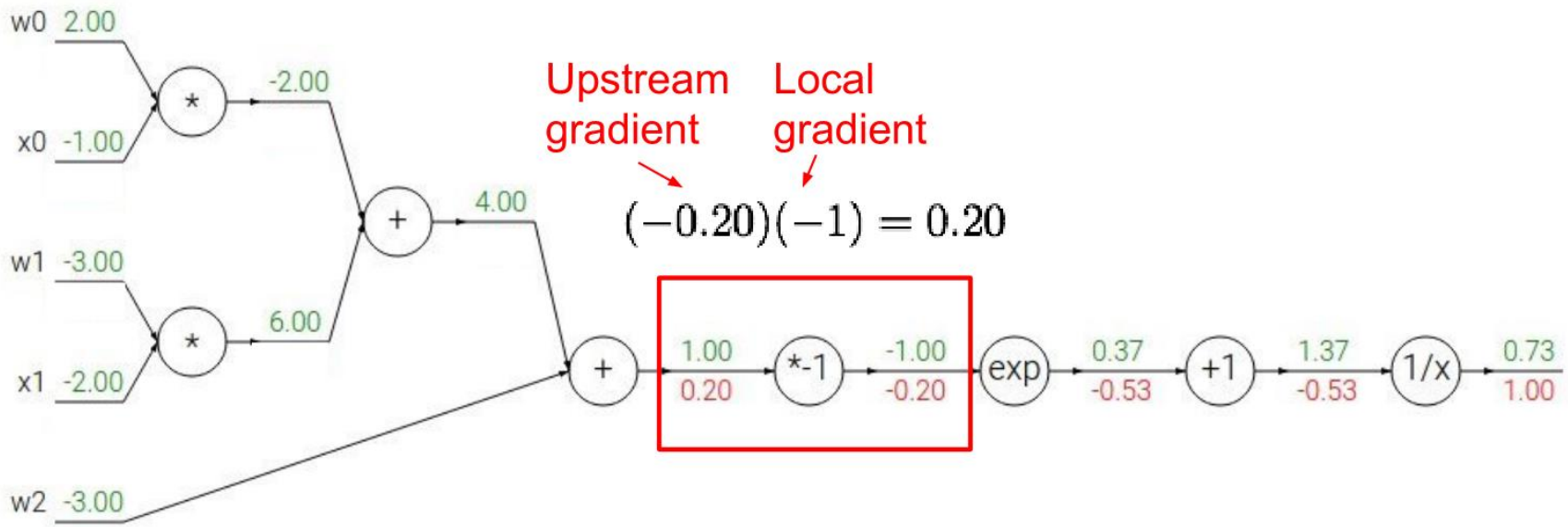
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Top-Down Calculation 4



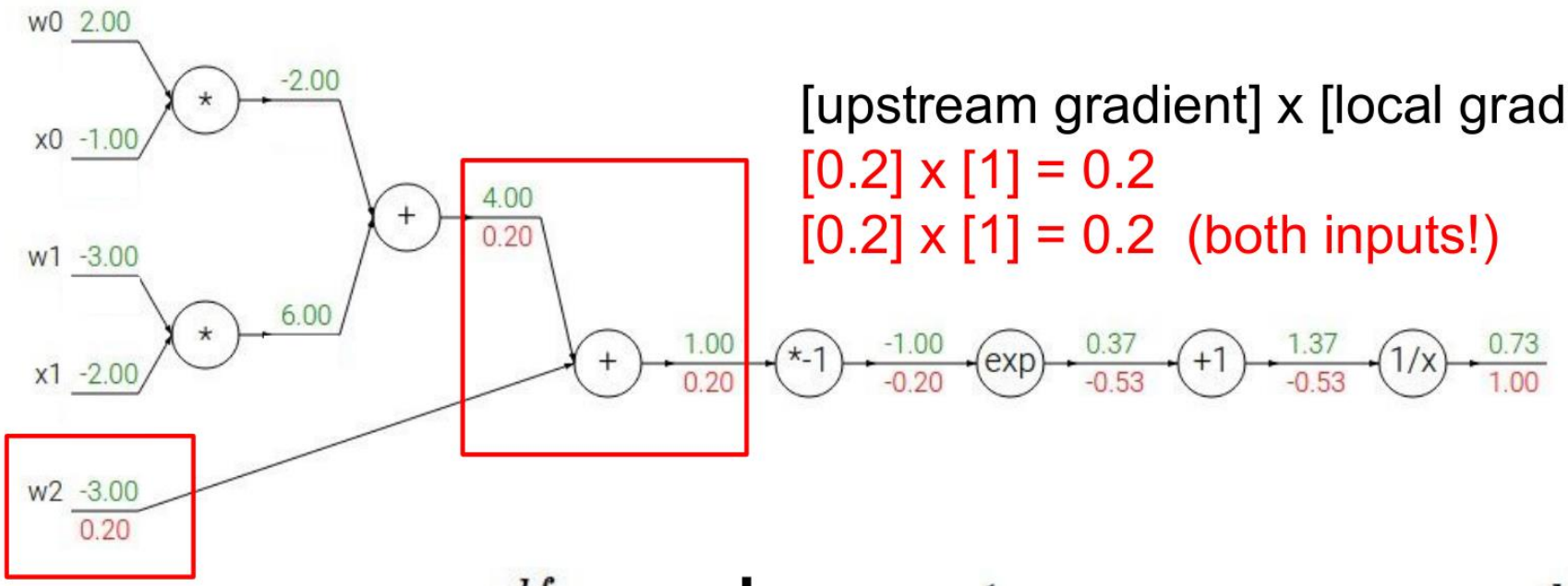
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Top-Down Calculation 5



[upstream gradient] x [local gradient]

[0.2] x [1] = 0.2

[0.2] x [1] = 0.2 (both inputs!)

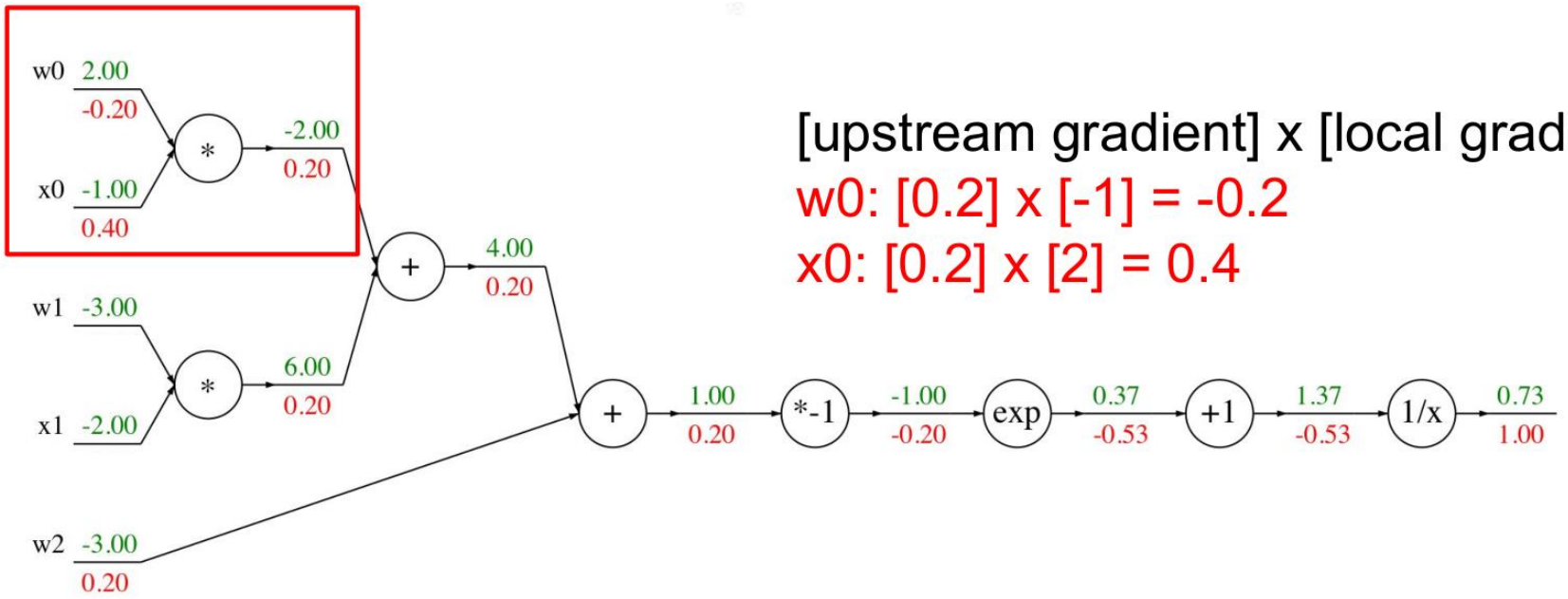
$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

Top-Down Calculation 6



[upstream gradient] x [local gradient]

w0: [0.2] x [-1] = -0.2

x0: [0.2] x [2] = 0.4

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Today’s Outlines

1. Stochastic Gradient Decent
2. Regularization

SGD

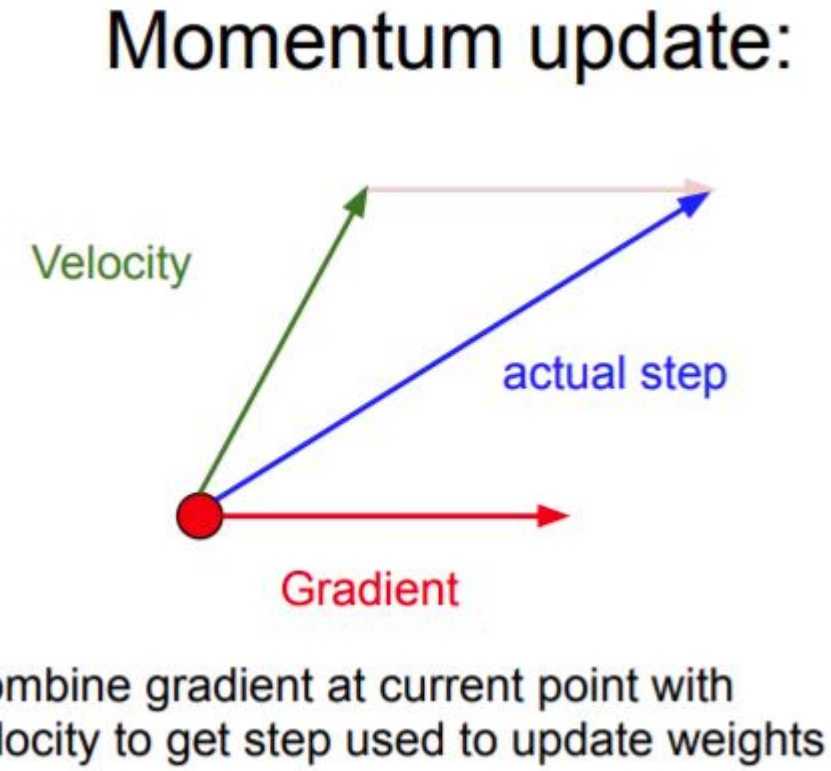
- Gradient estimates can be very noisy (Good and Bad)
- Use larger mini-batches to reduce stochastics
- Computation per update is efficient and does not depend on number of training samples.
  - Allows convergence on extremely large datasets

SGD + Momentum

- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations

$$g_{t+1} = \gamma g_t + \eta \nabla L_{t-1}$$

The momentum term γ is usually set to 0.9 or a similar value.



Comparison

- In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.

Image 2: SGD without momentum

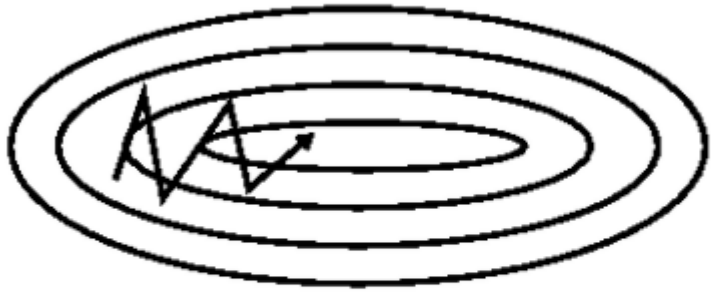


Image 3: SGD with momentum

The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$).

Adagrad (Adaptive Gradient Descent)

- Adagrad adapts the learning rate to the parameters,

- It performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features

$$w_{t+1,j} = w_{t,j} - \frac{\gamma}{\sqrt{G_{t,jj} + \epsilon}} \frac{\partial L}{\partial w_j}$$

where G_t is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. w_j up to time step t.

RMSprop

- RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

- We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $\mathrm{E}\left(\left[\frac{\partial L}{\partial w_j}\right]^2\right)$

$$w_{t+1,j} = w_{t,j} - \frac{\gamma}{\sqrt{\mathrm{E}\left(\left[\frac{\partial L}{\partial w_j}\right]^2\right) + \epsilon}} \frac{\partial L}{\partial w_j}$$

Examples

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(),
  metrics = c("accuracy")
)
```

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adagrad(),
  metrics = c("accuracy")
)

history <- model %>% fit(
  x_train, y_train,
  epochs = 20, batch_size = 128,
  validation_split = 0.2
)
```

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_sgd(),
  metrics = c("accuracy")
)

history <- model %>% fit(
  x_train, y_train,
  epochs = 20, batch_size = 128,
  validation_split = 0.2
)
```

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_sgd(momentum=0.9),
  metrics = c("accuracy")
)

history <- model %>% fit(
  x_train, y_train,
  epochs = 20, batch_size = 128,
  validation_split = 0.2
)
```

Regularization

- **L2 regularization** we add a component that will penalize large weights.

$$L_t(W) = \frac{1}{k_t - k_{t-1}} \sum_{i=k_{t-1}+1}^{k_t} L_i(x_i, y_i, W) + \frac{\lambda}{k_t - k_{t-1}} \left|\left|W\right|\right|^2_F$$

where λ is the **regularization parameter** and Frobenius norm is denoted by the subscript *F*.

Regularization

- **L1 regularization** we add a component that will penalize the sparsity of weights.

$$L_t(W) = \frac{1}{k_t - k_{t-1}} \sum_{i=k_{t-1}+1}^{k_t} L_i(x_i, y_i, W) + \frac{\lambda}{k_t - k_{t-1}} \left|\left|W\right|\right|^2_{L1}$$

where λ is the **regularization parameter** and L1 norm is denoted by the subscript *L1*.

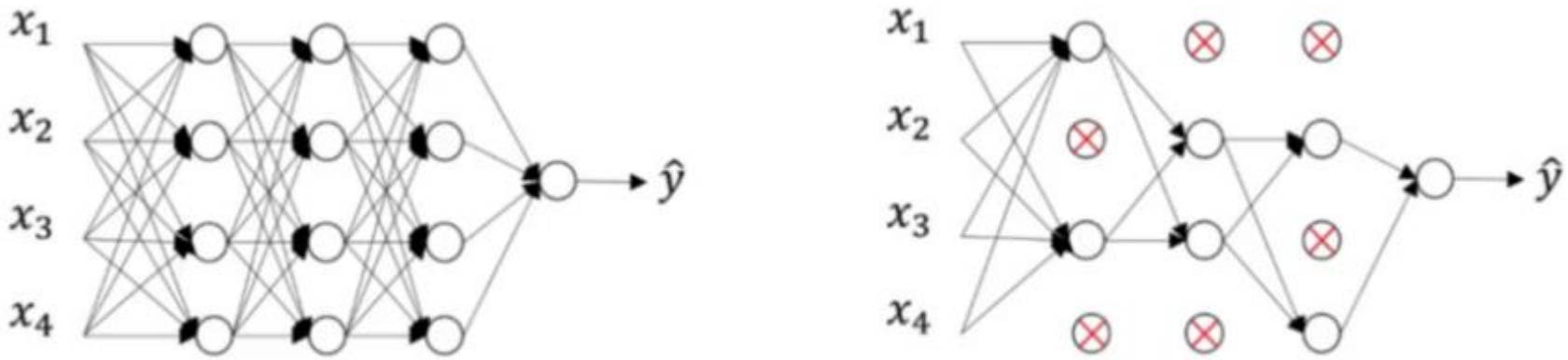
Implementation

```
# Use L1 regularizaition
```{r}
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = "relu", input_shape = c(784)
              ,kernel_regularizer = regularizer_l1_l2(l1 = 0.01)) %>%
  layer_dense(units = 128, activation = "relu",
              kernel regularizer = regularizer l1 l2(l1 = 0.01)) %>%
  layer_dense(units = 10, activation = "softmax")
```
```

```
# Use L2 regularizaition
```{r}
model <- keras_model_sequential()
model %>%
  layer dense(units = 256, activation = "relu", input shape = c(784)
              ,kernel_regularizer = regularizer_l1_l2(l2 = 0.01)) %>%
  layer_dense(units = 128, activation = "relu",
              kernel_regularizer = regularizer_l1_l2(l2 = 0.01)) %>%
  layer dense(units = 10, activation = "softmax")
```
```

Dropout regularization

- The probability of keeping each node is set at **random**. You only decide of the **threshold**: a value that will determine if the node is kept or not.
- For example, if you set the threshold to 0.7, then there is a probability of 30% that a node will be removed from the network.



Left: neural network before dropout. Right: neural network after dropout.

Why Dropout works

- Dropout means that the neural network cannot rely on any input node, since each have a random probability of being removed. Therefore, the neural network will be reluctant to give high weights to certain features, because they might disappear.

- Consequently, the weights are spread across all features, making them smaller. This effectively shrinks the model and regularizes it.

Implementation

```r
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = "relu", input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
```