

CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford



“Physics is the universe's operating system.” (Steven R Garman)

Final project

Goal

Implementing a neural network in order to recognize hand-written digits

Logistics

Turn in	Date	Grade
Preliminary report + code	Friday May 28th	20%
Final report (4 pages) + code	Sunday June 6th	80%

Preliminary report

Focus is on correctness

Final report

Profiling and analysis, performance, quality of report

What are the performance bottlenecks in your code?

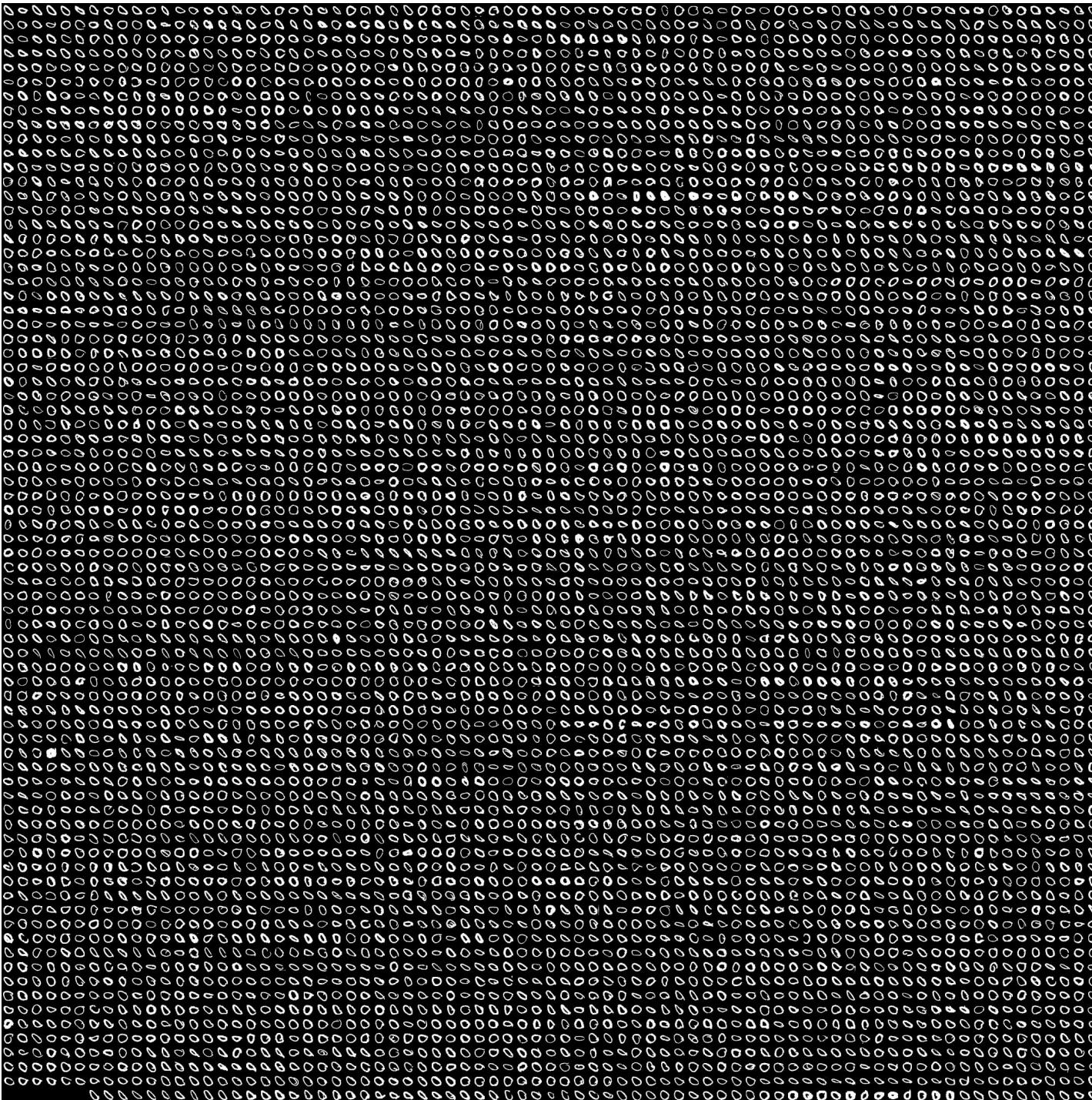
How can they be addressed?

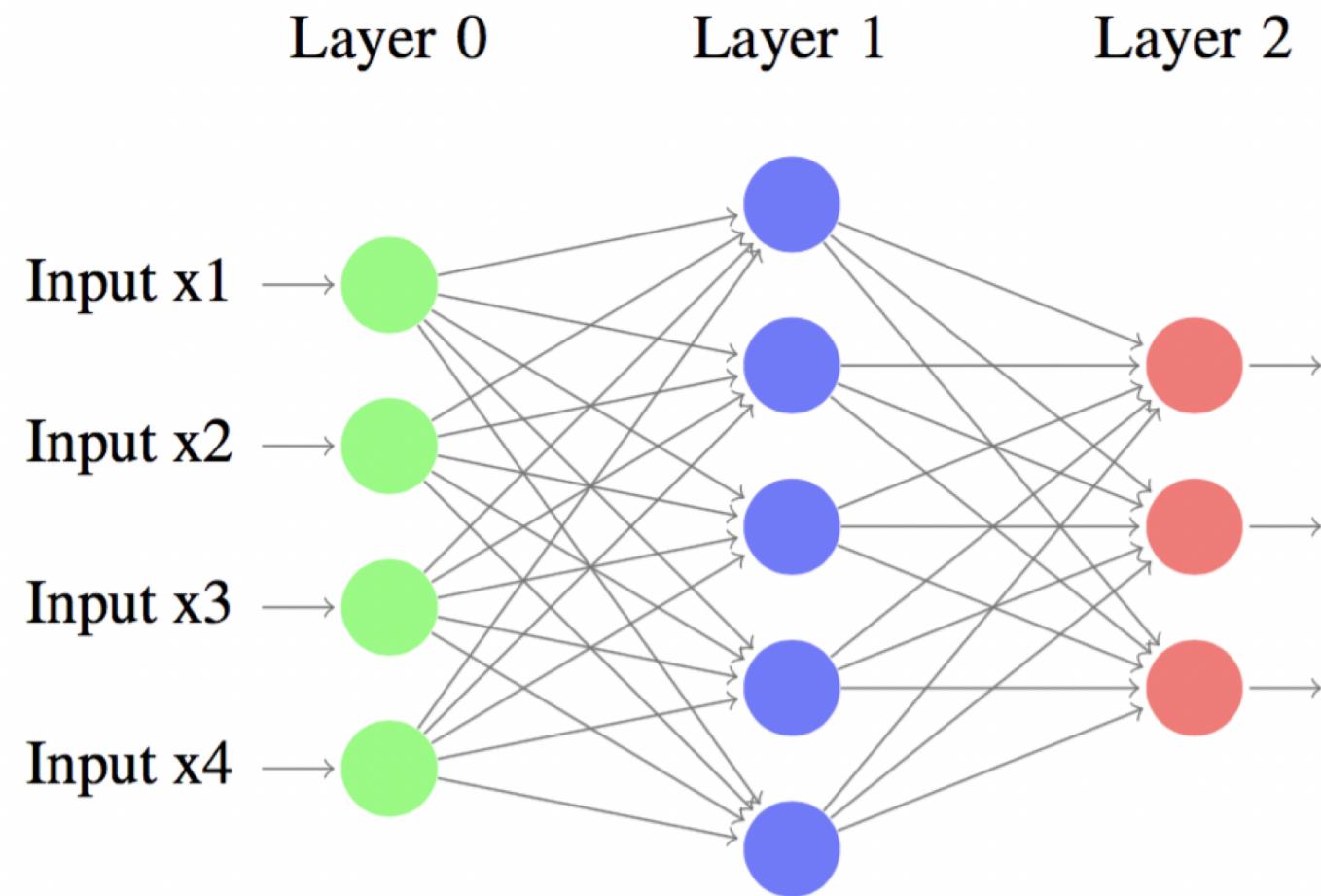
Correctness

Discuss your strategy to test your code

Test outputs for valid inputs

Make sure you distinguish roundoff errors from genuine bugs





Input layer: image

Hidden layer: -n num; variable size

Output layer: softmax vector with 10 digits

Softmax

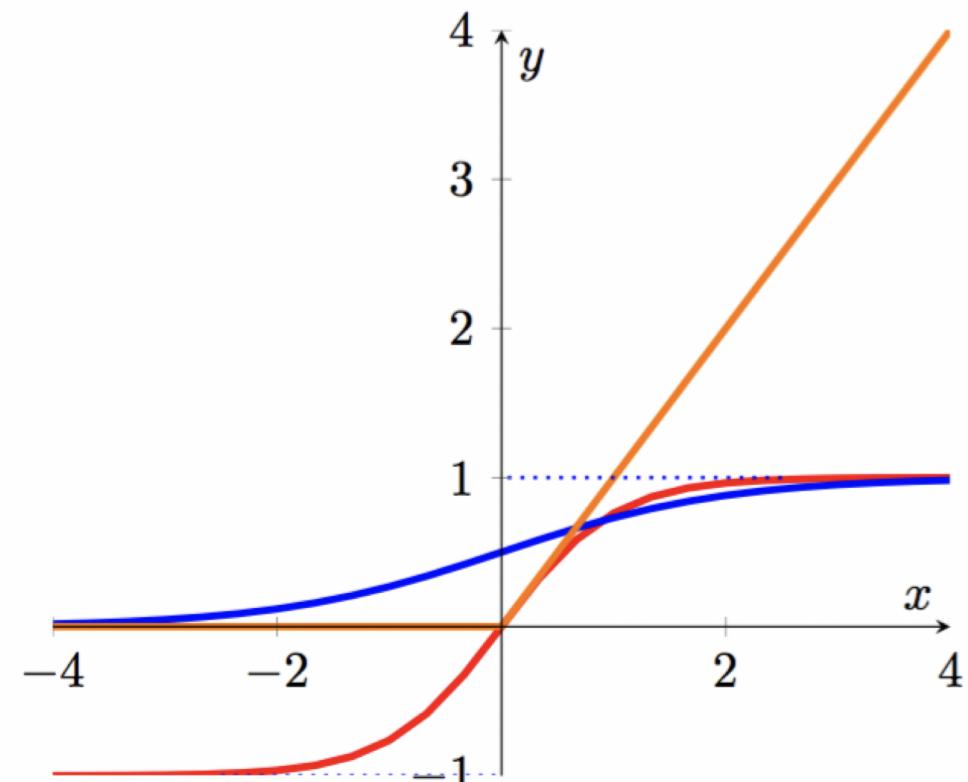
$$\text{softmax}(z)_j = \frac{\exp(z_j)}{\sum_{i=0}^9 \exp(z_i)}$$

Interpreted as a probability

Each layer is a matrix multiplication and a non-linear function

$$z = Wx + b$$

$$a = \sigma(z)$$



— $\tanh(x)$ — Sigmoid — ReLU

We will use sigmoid

How do you train a network?

Many methods but most are based on gradient descent

Error function

$$J(p) = \frac{1}{N} \sum_{i=1}^N \text{error}^{(i)}(y_i, \hat{y}_i)$$

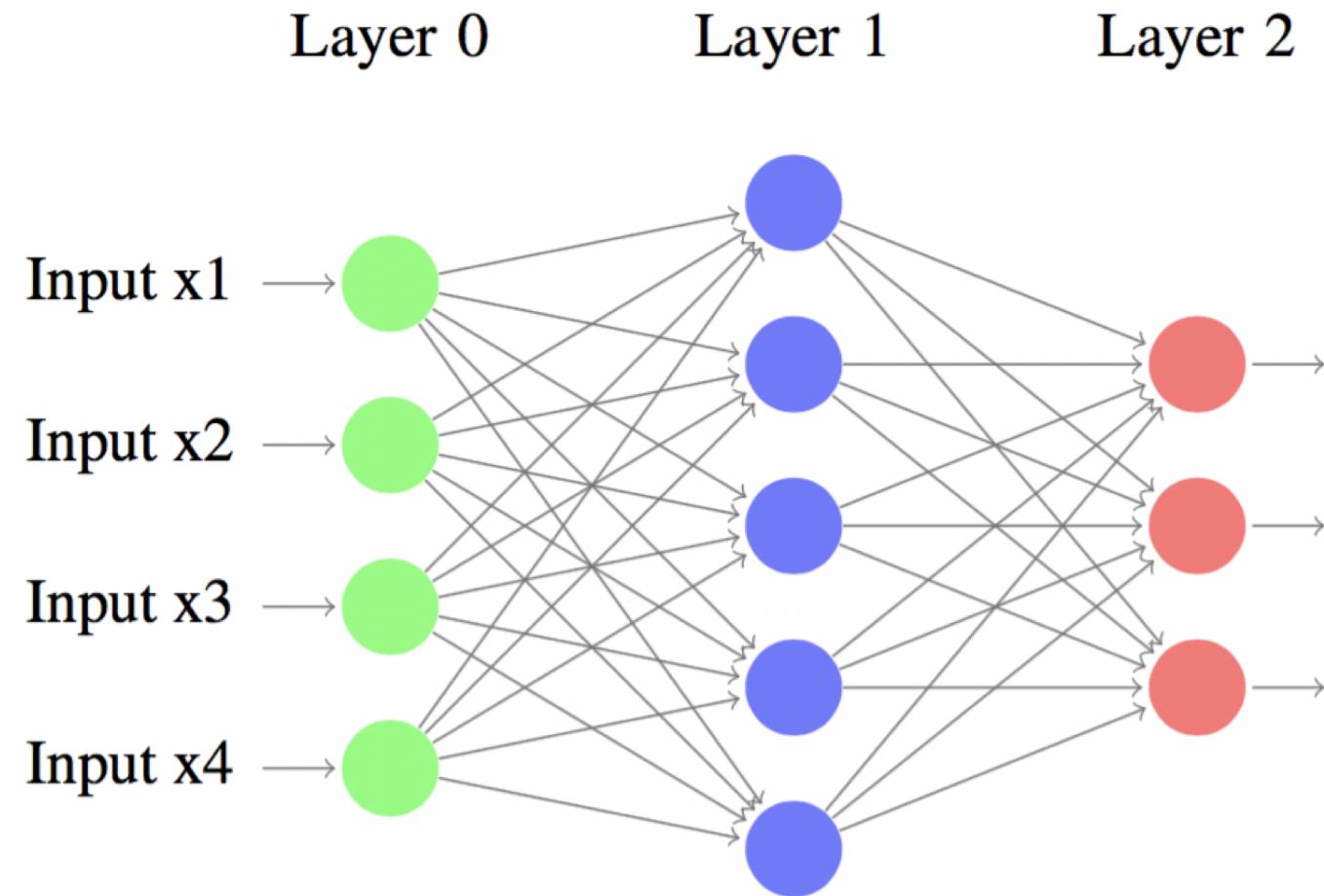
p : weights and biases of network = all parameters

Gradient update

$$p \leftarrow p - \alpha \nabla_p J$$

Gradient is computed by repeated application of the chain rule

Backpropagation



Stochastic gradient descent

$$J_r(p) = \frac{1}{N} \sum_{i \in \text{random subset}} \text{error}^{(i)}(y_i, \hat{y}_i)$$

$$p \leftarrow p - \alpha \nabla_p J_r$$

If we use a small subset, this allows more updates to the DNN coefficients
⇒ more accurate

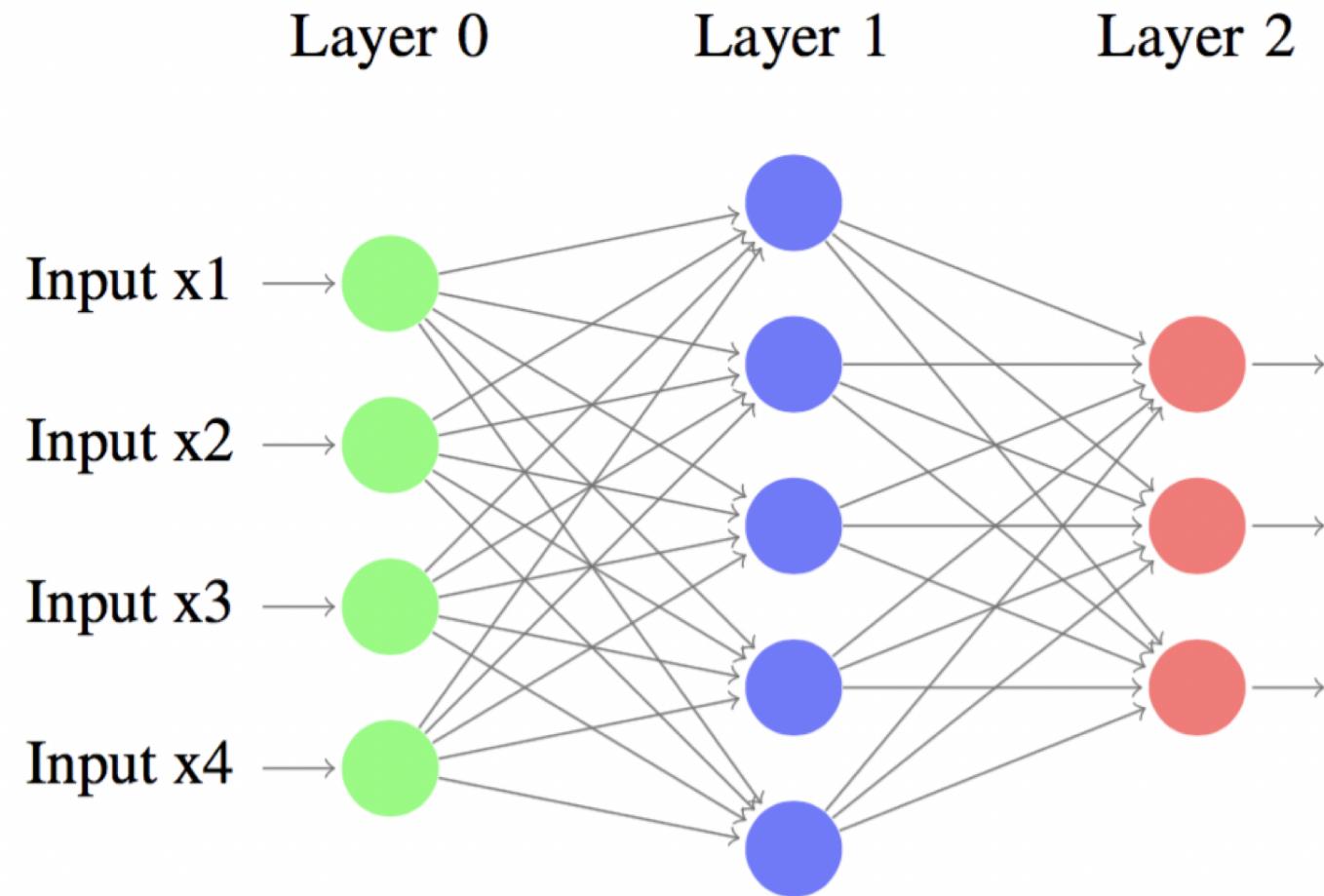
Randomness of subset selection allows avoiding local minima and escaping saddle points
⇒ better convergence

Sequence of operations

Forward pass = left to right; DNN prediction; compare with label

Backward propagation = right to left; chain rule; compute gradient and update DNN

Iterate until convergence



Core building blocks to implement

- Matrix-matrix products
- Non-linear activation functions

<https://playground.tensorflow.org>



Epoch
000,693

Learning rate
0.03

Activation
Tanh

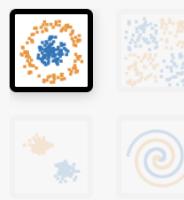
Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

- x_1
- x_2
- x_1^2
- x_2^2
- $x_1 x_2$
- $\sin(x_1)$
- $\sin(x_2)$

+

-

2 HIDDEN LAYERS

+

-

4 neurons

2 neurons

+



Epoch
000,259

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 50%

Noise: 0

Batch size: 10

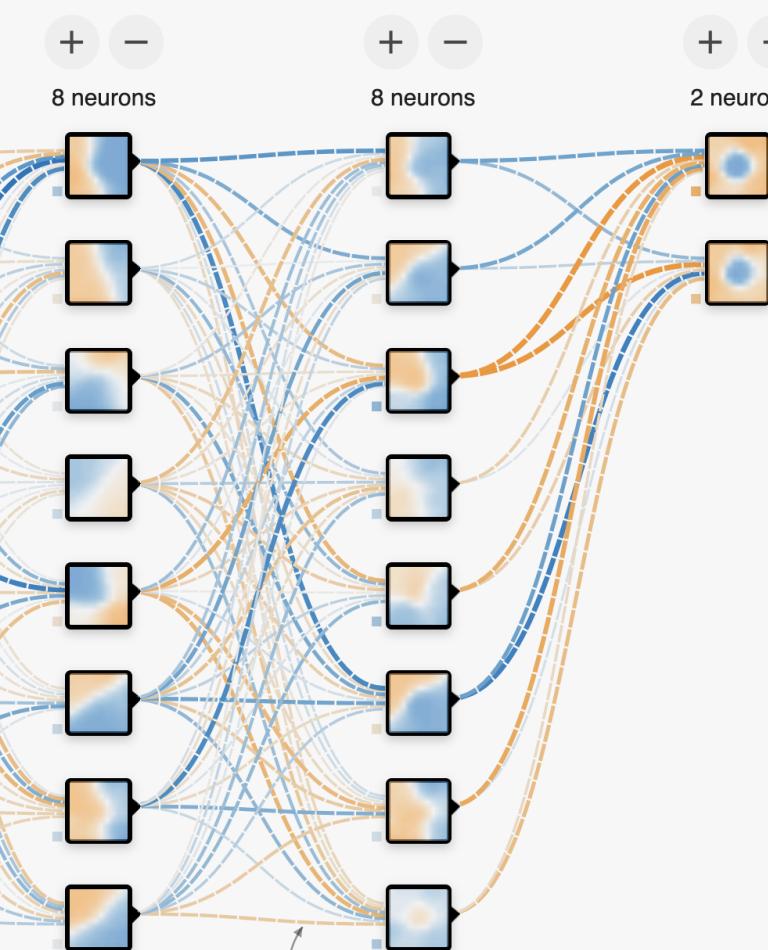
REGENERATE

FEATURES

Which properties do you want to feed in?

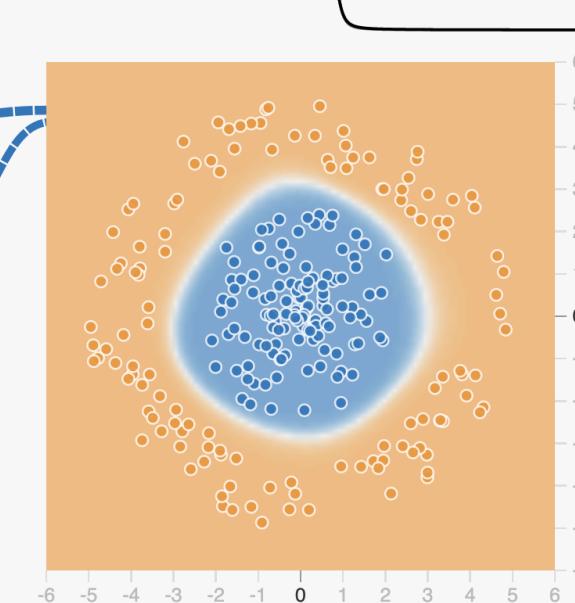
- x_1
- x_2
- x_1^2
- x_2^2
- x_1x_2
- $\sin(x_1)$
- $\sin(x_2)$

4 HIDDEN LAYERS

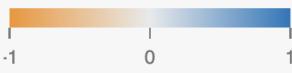


OUTPUT

Test loss 0.001
Training loss 0.000



Colors shows
data, neuron and
weight values.

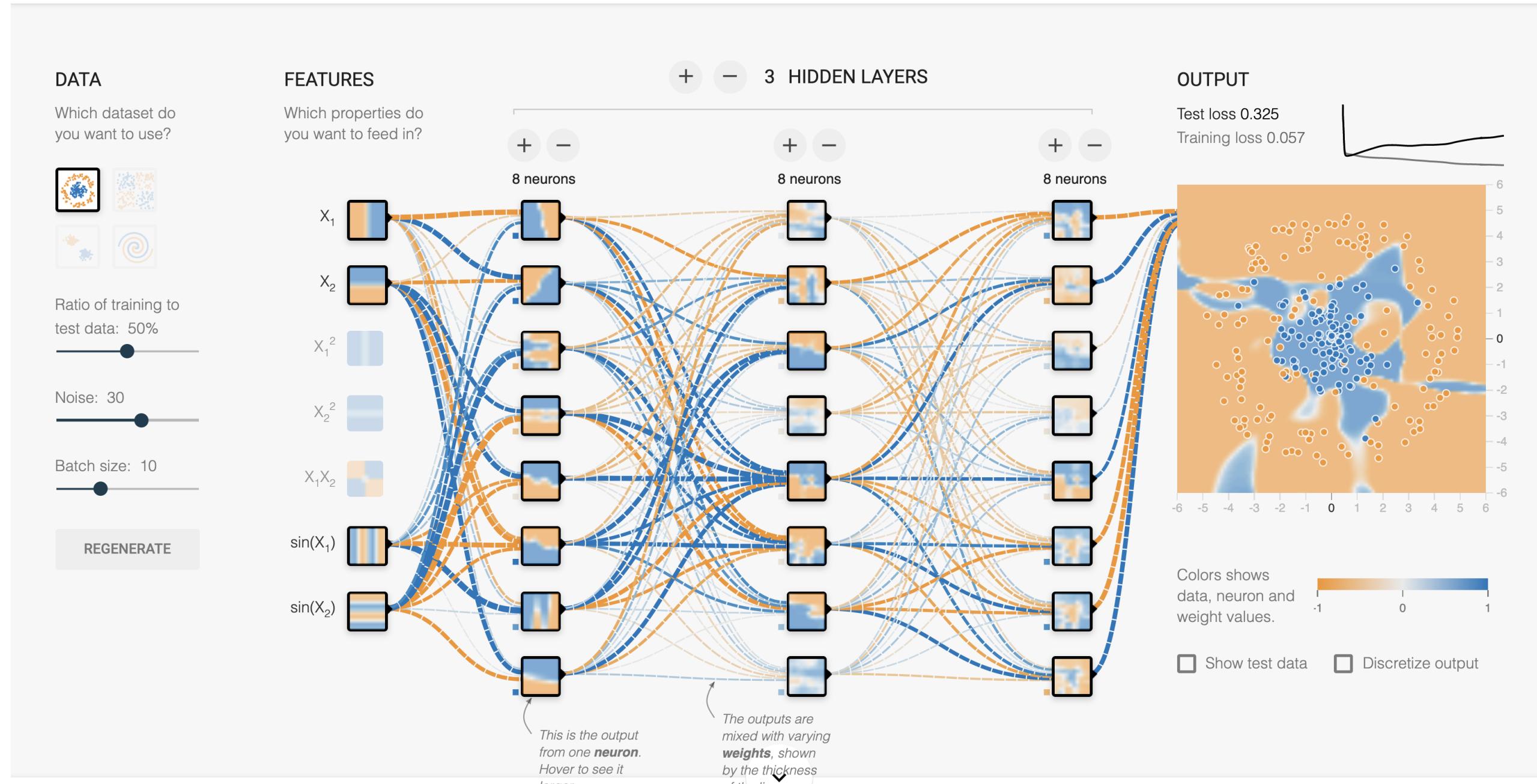


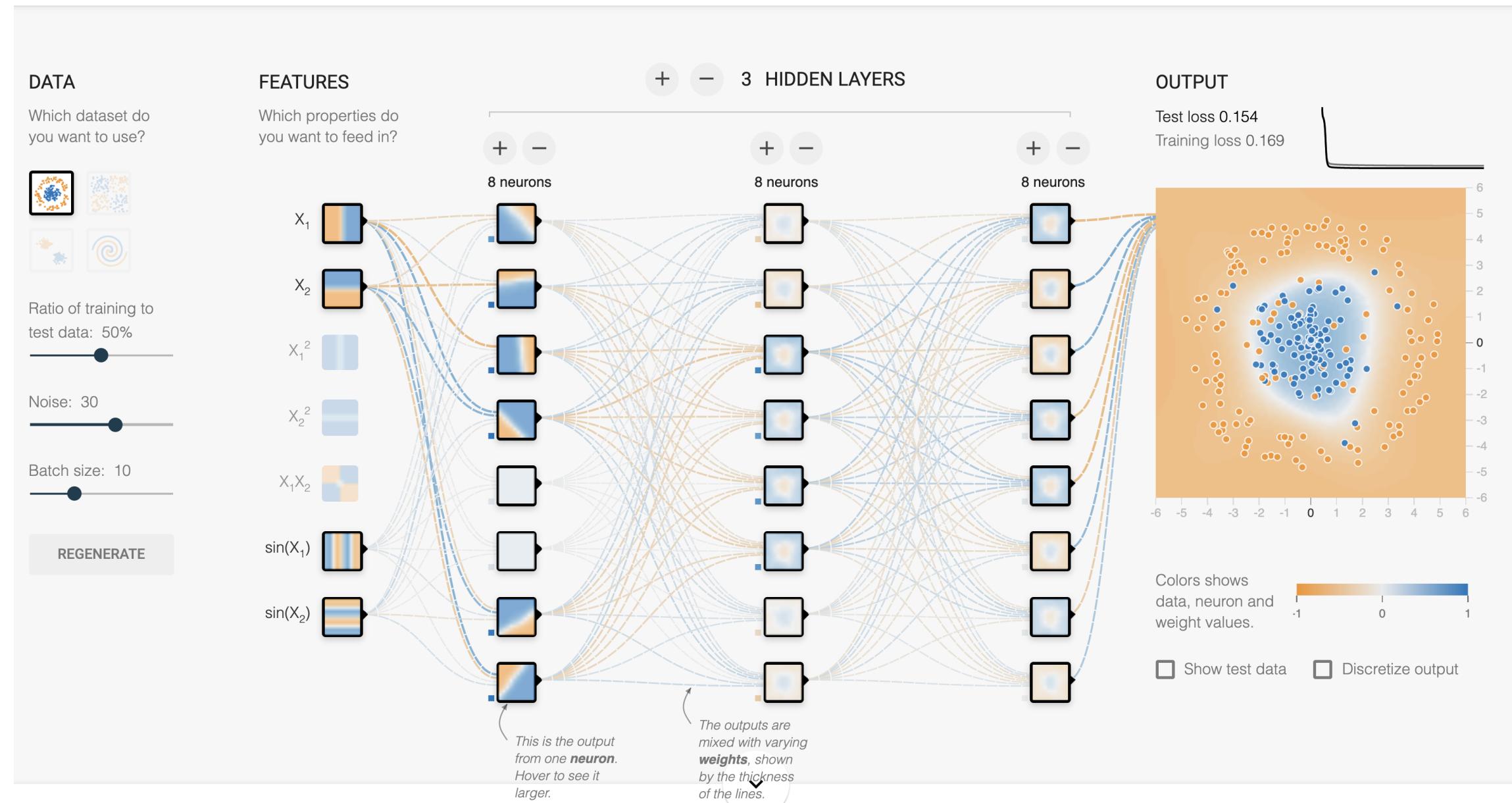
This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Show test data

Discretize output

Epoch
001,298Learning rate
0.03Activation
TanhRegularization
NoneRegularization rate
0Problem type
Classification

Epoch
000,946Learning rate
0.03Activation
TanhRegularization
L2Regularization rate
0.03Problem type
Classification

Regularization

$$J(p) = \frac{1}{N} \sum_{i=1}^N \text{error}^{(i)}(y_i, \hat{y}_i) + \frac{1}{2} \lambda ||p||_2^2$$

p : weights and biases of the network

$$z = Wx + b$$

$$a = \sigma(z)$$

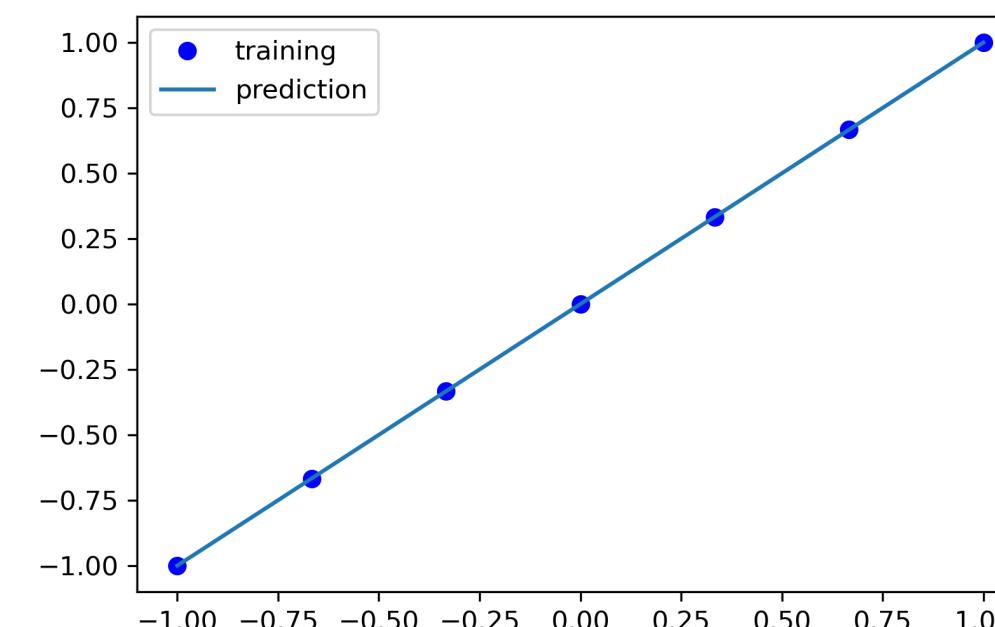
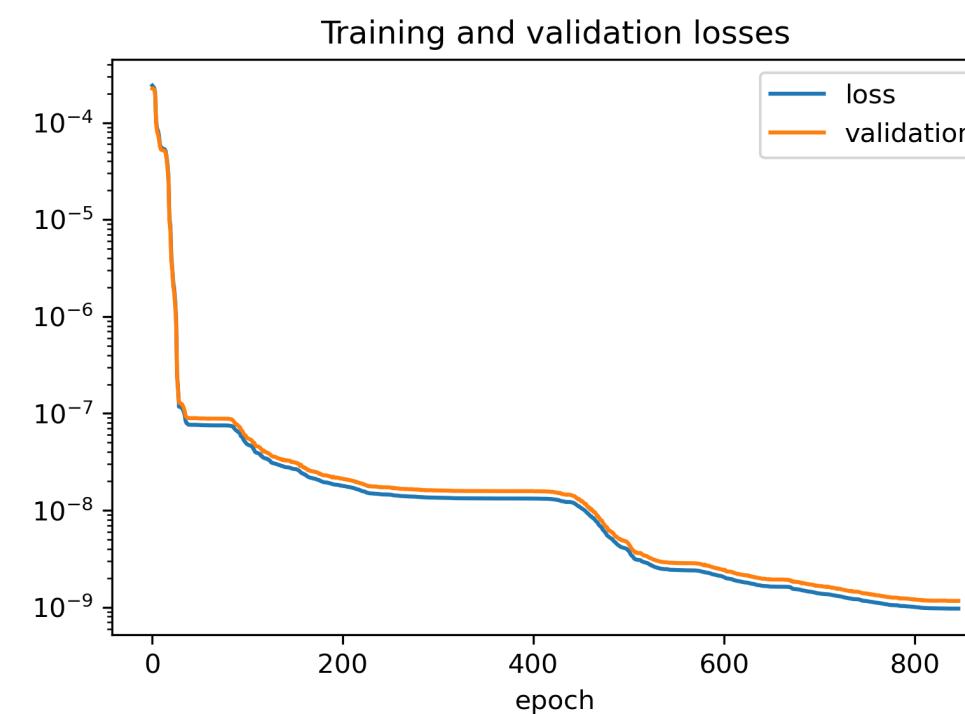
Regularization makes the DNN more linear

How can we figure out how much regularization is needed?

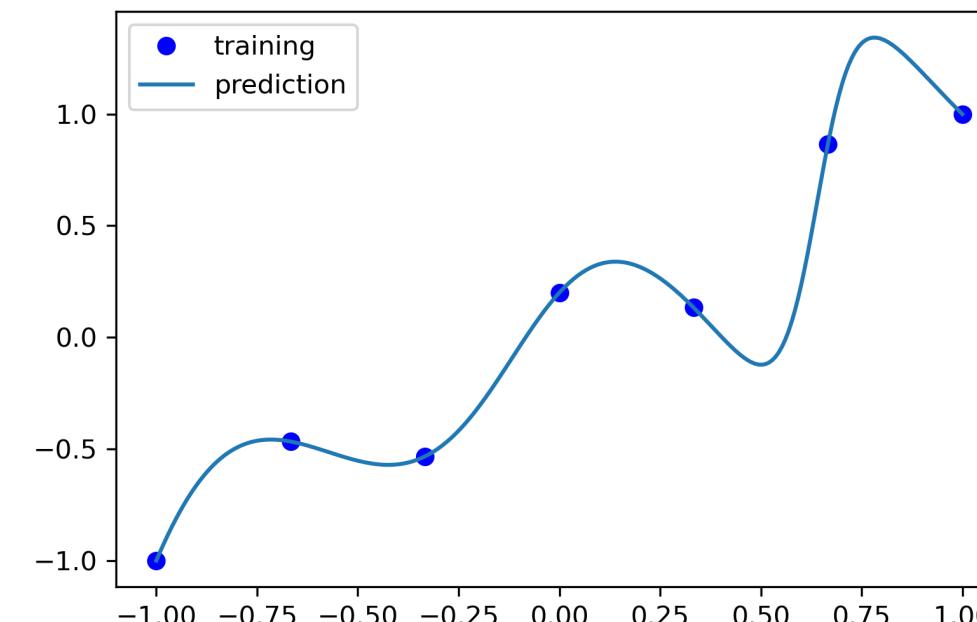
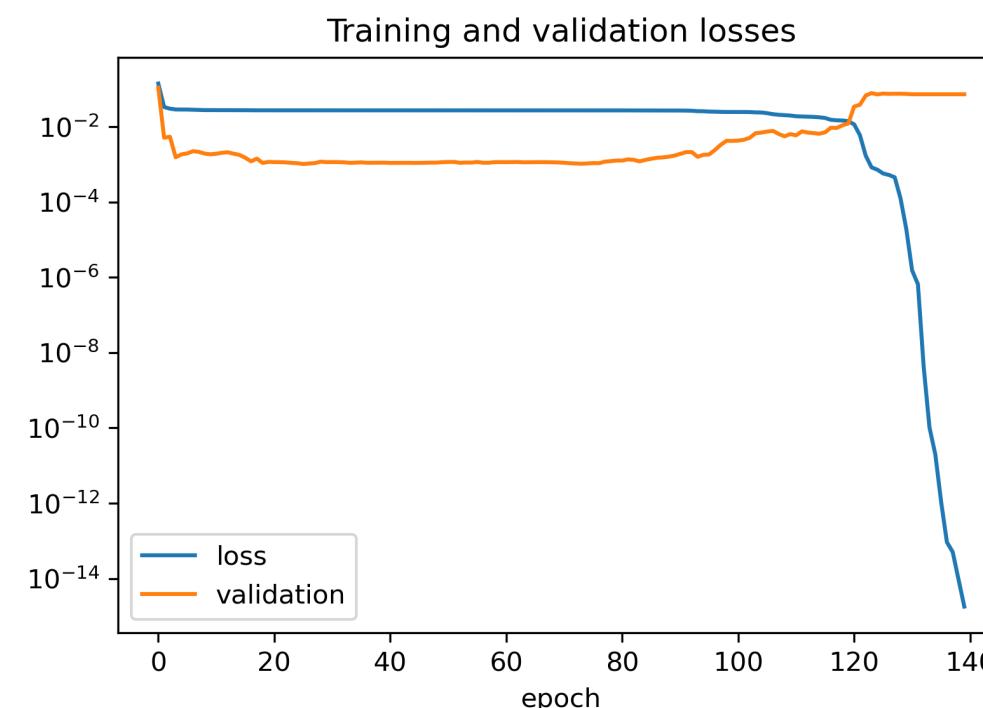
Training set: used to minimize loss; involved in defining the gradient

Validation set: used to evaluate model; how accurate is it? Avoids overfitting

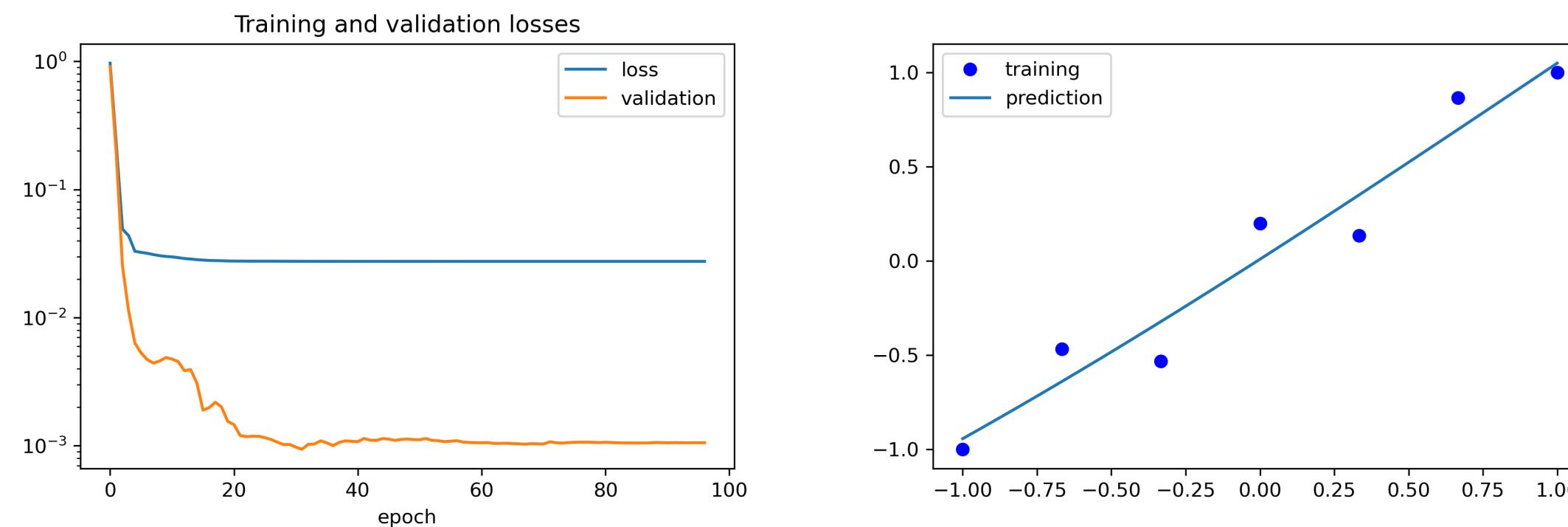
Example: small 2-layer DNN with width 8



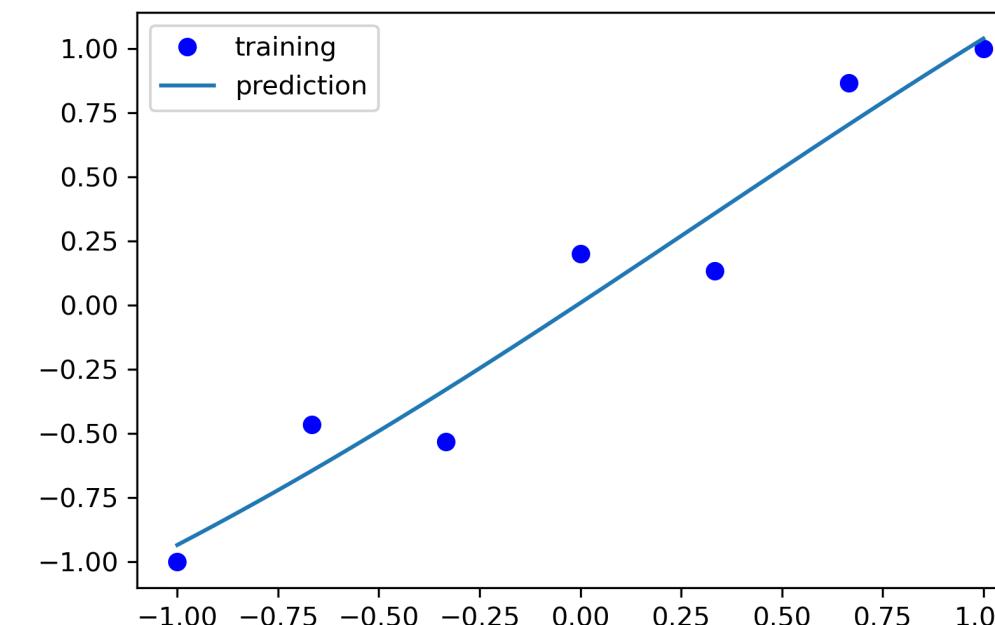
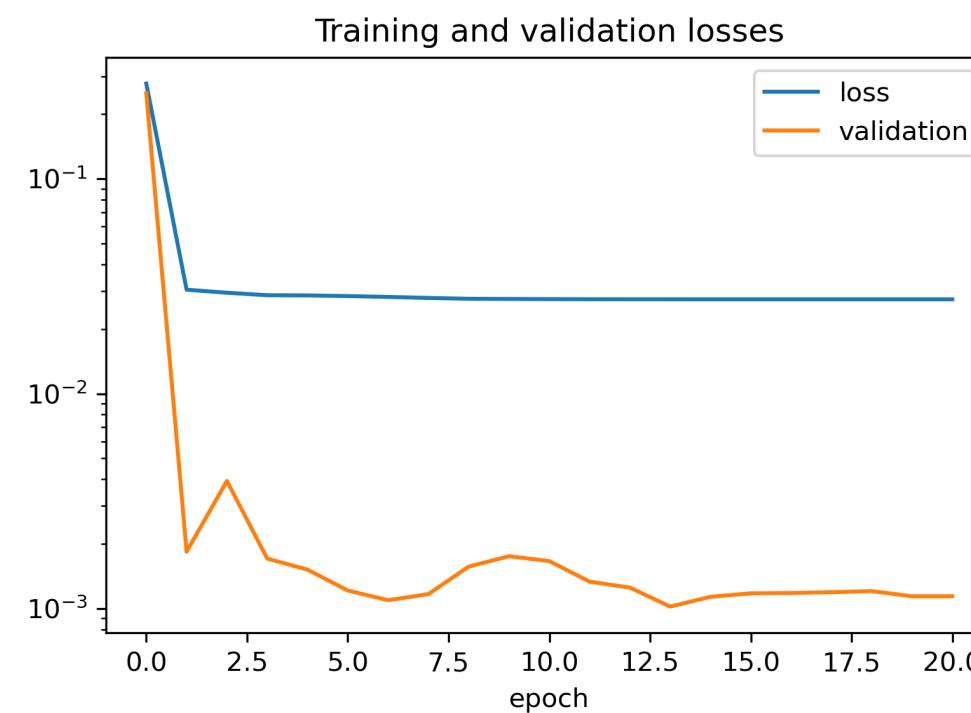
With noise added to data



Fix 1: reduce the size of the DNN; for example with width 1



Fix 2: add regularization, e.g., $\lambda = 10^{-3}$



Training



Validation



Diagnostic

Overfitting; $\uparrow \lambda$

Too much regularization;
 $\downarrow \lambda$

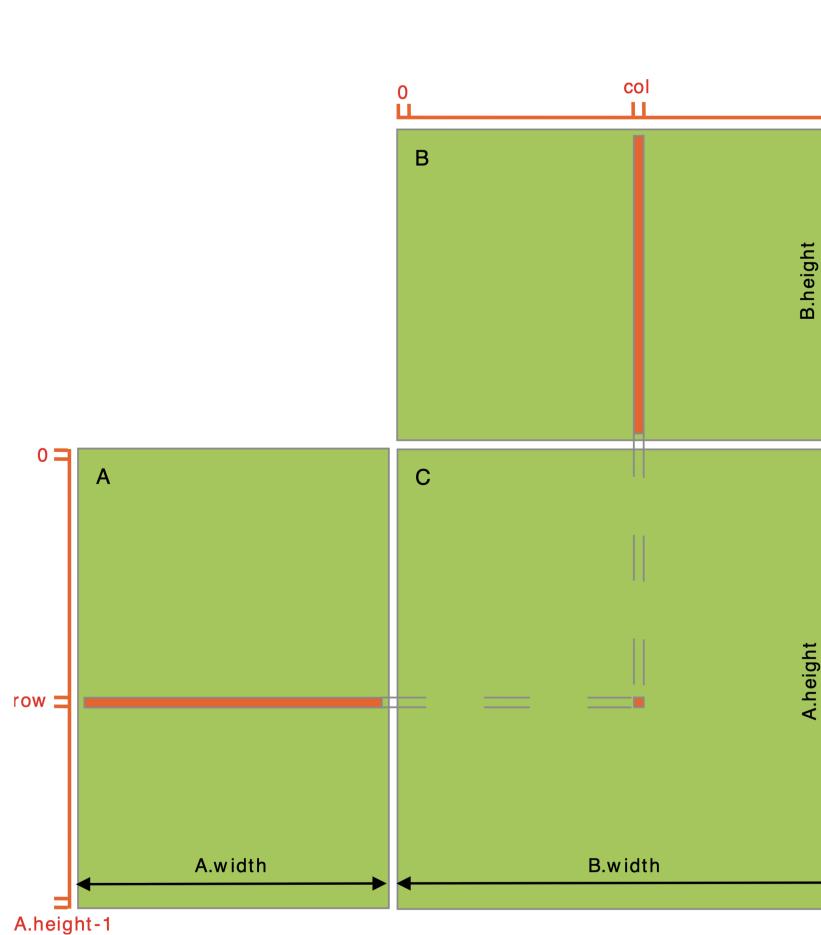
Regularization is good

- Training set: optimize DNN parameters
- Validation set: optimize regularization

Two main tasks in the project

1. Implement a matrix-matrix product (GEMM) algorithm
2. Implement the MPI algorithm for distributed memory

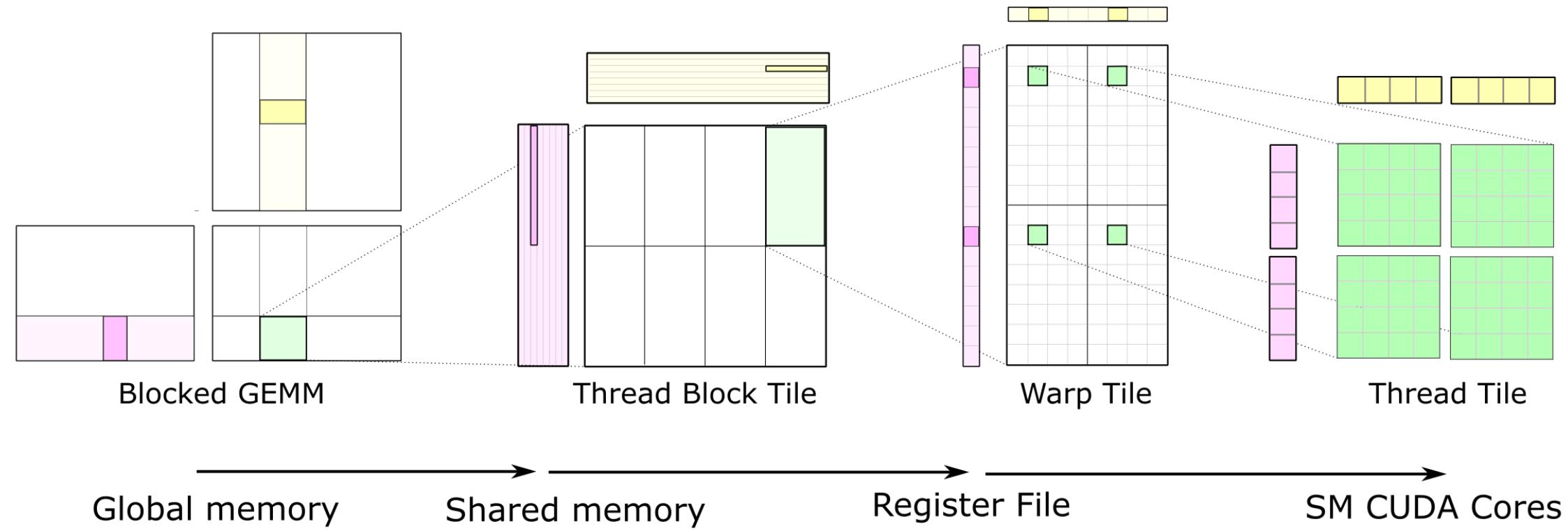
Naive implementation; shared memory is not used



GEMM performance

The key is to increase the arithmetic intensity.

This requires reducing the memory traffic.



$$c_{ij} = \sum_k a_{ik} b_{kj}$$

$$c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$$

$$c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$$

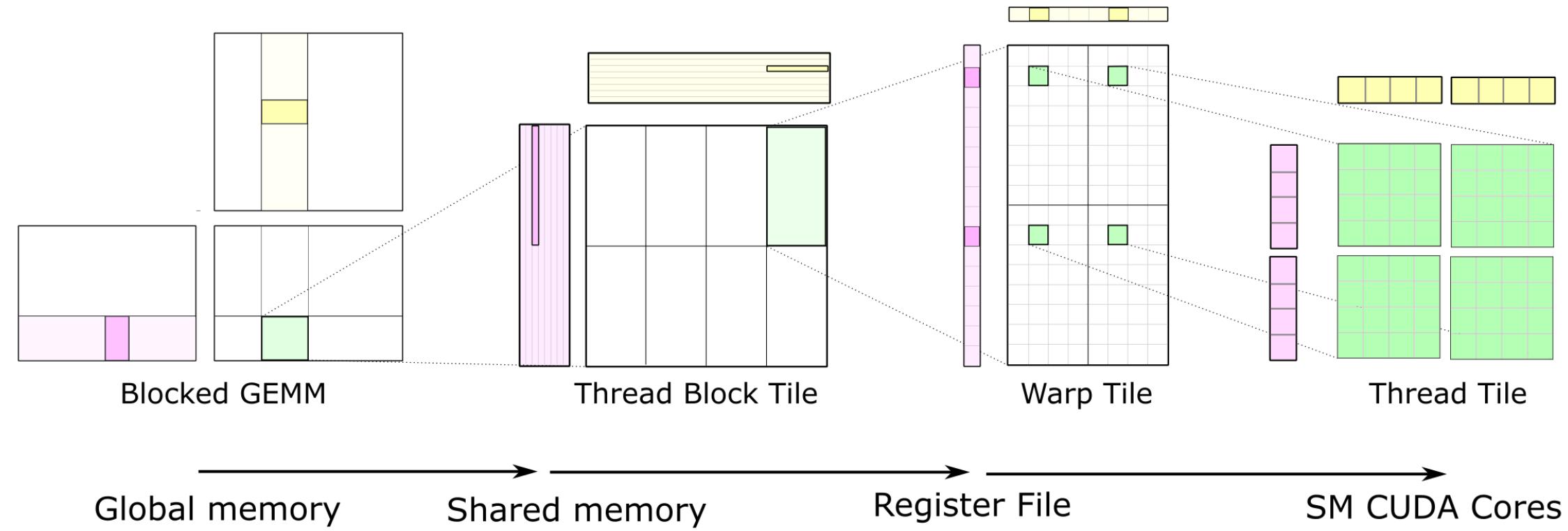
Block size: b

Memory traffic: $2b$

Flops: b^2

High arithmetic intensity



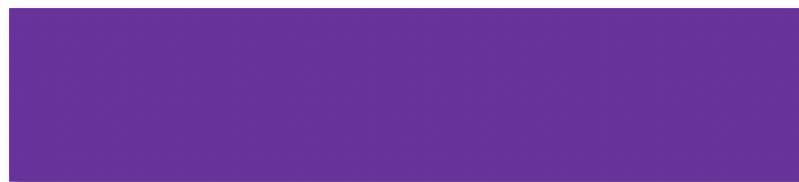


MPI, distributed memory algorithm

Topic of upcoming lectures

High-level discussion of approaches

Output



Layer 1

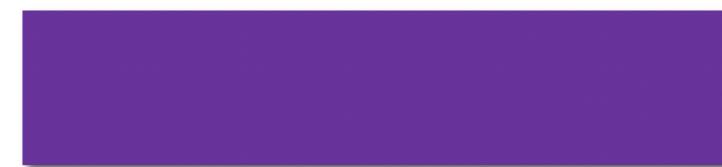


Image



Data parallelism

Output



Layer 1



Image



Communication

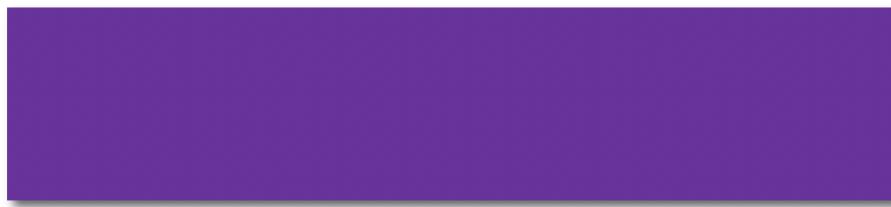
$$J(p) = \frac{1}{N} \sum_{i=1}^N \text{error}^{(i)}(y_i, \hat{y}_i)$$

Sum is required over all input images to compute gradient

Parallel reduction to get $\nabla_p J$

= Reduction for all DNN coefficients across all nodes

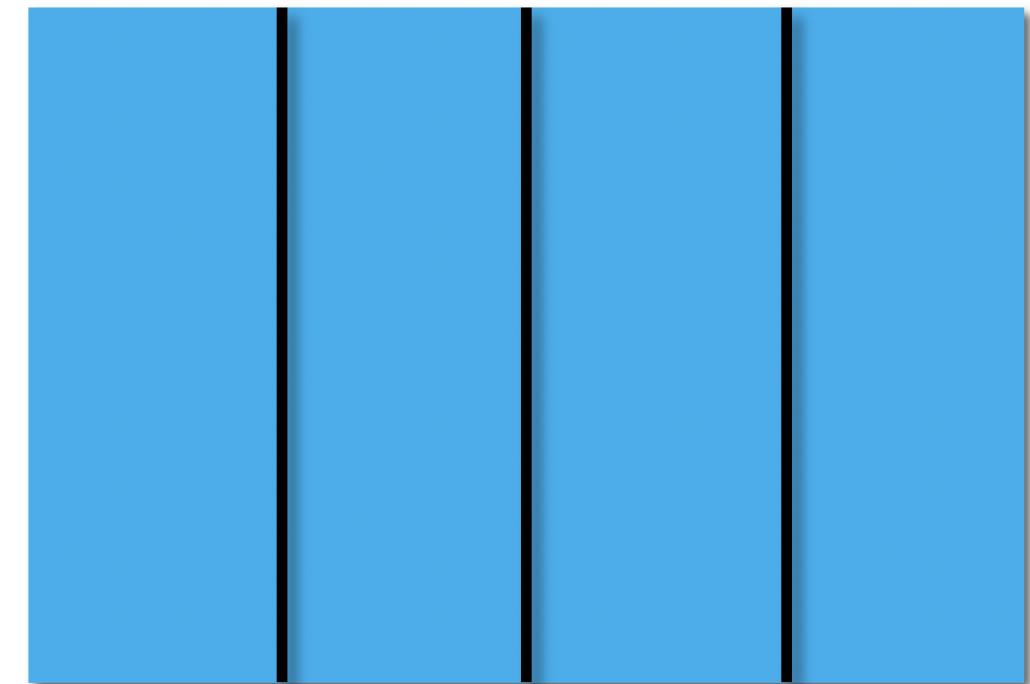
Output



Layer 1



Image

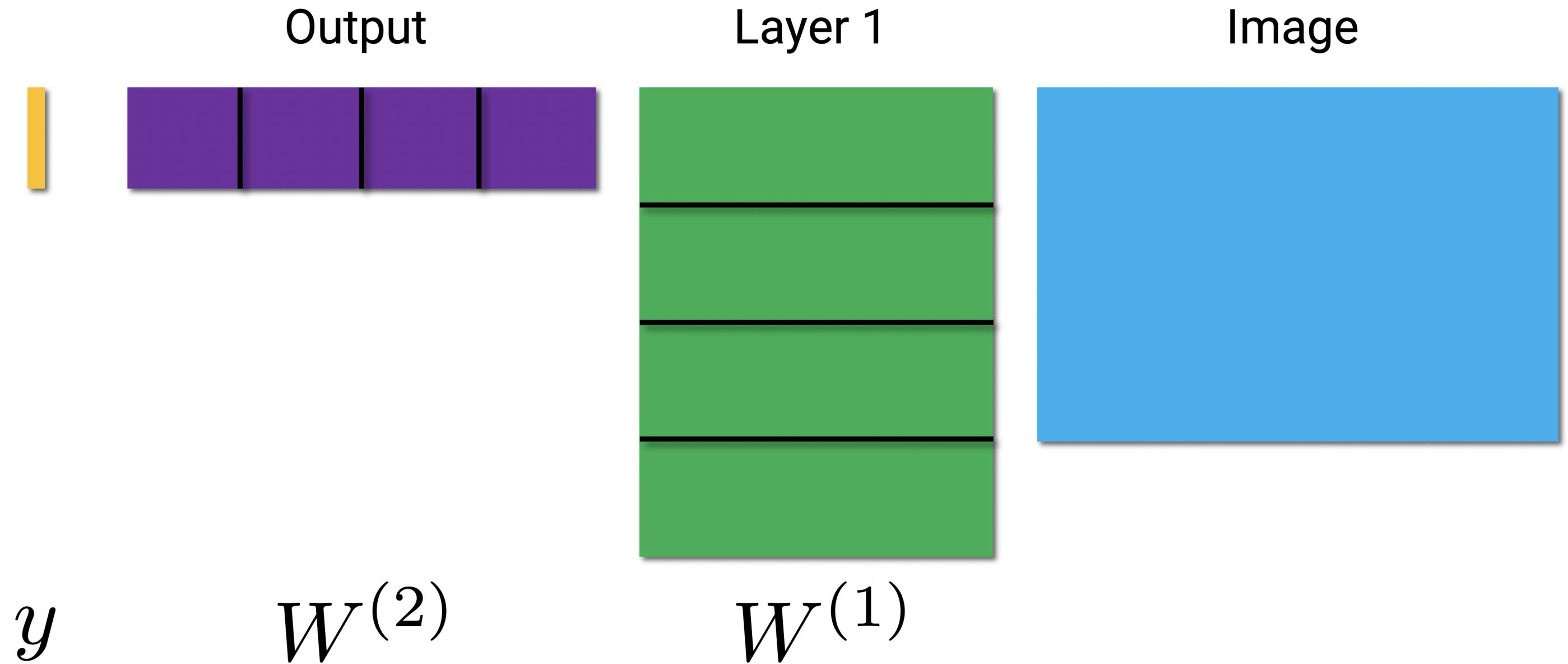


Time for MPI communication is fairly significant.

A better implementation exists!

Model parallelism

Much more complicated to understand but implementation is not more difficult than previous approach



Reduction is required at the end to get the output labels y

Backpropagation

You have not seen the details yet. So, it will be hard to follow.

The take-home message is that no MPI communication is required between nodes.



$$[W^{(2)}]^T y$$

$$W^{(1)} \leftarrow W^{(1)} - \alpha [(W^{(2)})^T y] x^T$$

Warning!

Equations in previous slide were simplified for clarity

See Part1 write-up for details

No communication is required during the backpropagation

This implementation is much more efficient

