

Lecture 14

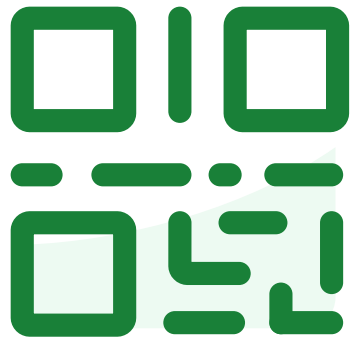
PyTorch, Initialization, and Normalization

Tensors, autograd, modules, and training loops in PyTorch. Parameter initialization, batch normalization, and layer normalization.

EECS 189/289, Fall 2026 @ UC Berkeley

Joseph E. Gonzalez and Narges Norouzi

Do not edit
How to change the
design



**Join at slido.com
#1809914**

① The Slido app must be installed on every computer you're presenting from

slido

Roadmap for Today

- PyTorch
- Parameter Initialization
- Normalization



1809914



1809914

The most widely used **framework** for **building** and **training** deep **neural networks**.

- *Related frameworks:* [JaX](#), [TensorFlow](#), ...

Important features:

- **Automatic differentiation** (covered in Lecture 13)
- Intuitive support for **hardware accelerators**
- **Easy to use APIs** for state-of-the-art **model architectures**

Learning on your own:

- Basics: <https://docs.pytorch.org/tutorials/beginner/basics>
- Tutorials: <https://docs.pytorch.org/tutorials/>



1809914

Big Concepts in PyTorch

- **torch.tensor** is the basic datatype used to encode data, weights, and activations; it is very similar to **numpy.ndarrays**.
- Neural networks are built using the many primitives in the **torch.nn** module and defining the **forward** process of making a prediction.
- PyTorch uses automatic differentiation to efficiently compute the **gradient (already covered in Lecture 13)**.
- Training is accomplished by **(you) implementing** some variation of gradient descent.
 - Doesn't have a built in 'fit' function like scikit-learn



1809914

torch.tensor

The basic building block of PyTorch is the **torch.tensor** which is very similar to the **numpy.ndarray**.

- **Construction:** `torch.tensor([[1.0], [2.0]]); torch.ones(2,3)`
- **Indexing and slicing:** `x[:, 3]`
- **Shape and type:** `x.shape; x.dtype`
- **Operations:** `torch.mean(x, dim=1); a.T @ x`

Key differences:

- Greater emphasis on **dtype** (often `torch.bfloat16`, `torch.float16`)
- **Device Placement:** `x.to("cuda"); x.to("cpu")`
- Tensors (can) track how they were computed to enable **automatic differentiation:** `x.grad_fn`



1809914

torch.nn.functional and torch.nn

The **torch.nn.functional** Python module contains the basic **stateless** neural network functions (e.g., relu).

The **torch.nn** Python module contains the **basic building blocks** of neural networks.

- **nn.Module** is the base class which all models and layers extend.
 - **__init__**: defines all the model parameters.
 - **forward** (**__call__**): defines the model/layer logic
- **nn.Parameter** is used to define model parameters (in **__init__**)
 - Used to wrap parameter tensors so they can be automatically tracked
- There are hundreds of layer modules: **nn.Linear**; **nn.Softmax**; ...



1809914

torch.nn.Module

When building a new model you extend the nn.Module

```
class MLP(nn.Module):
```

Extend nn.Module

```
    def __init__(self, dims, activation="relu"):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(dims[i], dims[i+1]) for i in range(len(dims)-1)])
        if activation == "relu": self.act = nn.functional.relu
        else: self.act = torch.tanh

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i < len(self.layers)-1:
                x = self.act(x)
        return x
```

Add modules to your module. These add parameters.

Compute the **forward** pass using the layer modules.



1809914

Computing Gradients using Autograd

PyTorch uses **automatic differentiation** to **automatically** (and **efficiently**) **compute the gradient** of the loss with respect to all parameters.

```
model.zero_grad()
```

Reset the Gradient
(gradients
accumulate)

```
pred = model(x)
```

```
loss = loss_fn(pred, t)
```

Compute the loss

```
loss.backward()
```

Compute the **gradient** of the loss

```
print(model.layers[0].weight.grad)
```

Gradients are attached
to each parameter



1809914

Training Loop with an Optimizer

In PyTorch you implement **your own gradient descent loop** but you can use built in update rules (e.g., Adam) called **optimizers**.

```
loader = DataLoader(train_data, batch_size=32, shuffle=True)
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=eta)
```

```
for epoch in range(n_epochs):
```

```
    for x, t in loader:
```

```
        optimizer.zero_grad()
```

```
        loss = loss_fn(model(x), t)
```

```
        loss.backward()
```

```
        optimizer.step()
```

Minibatch
SGD
DataLoader

Adam
optimizer

Epoch is one pass over data

Load one minibatch at a time

Compute the gradient
of the loss

Use optimizer to update all the parameters (1-step)



1809914

Demo

Notebook, Part 1: tensors, autograd, data loaders, modules, losses, and the training loop.

Roadmap

- PyTorch
- Parameter Initialization
- Normalization



1809914

Choice of Initial Parameters Matters

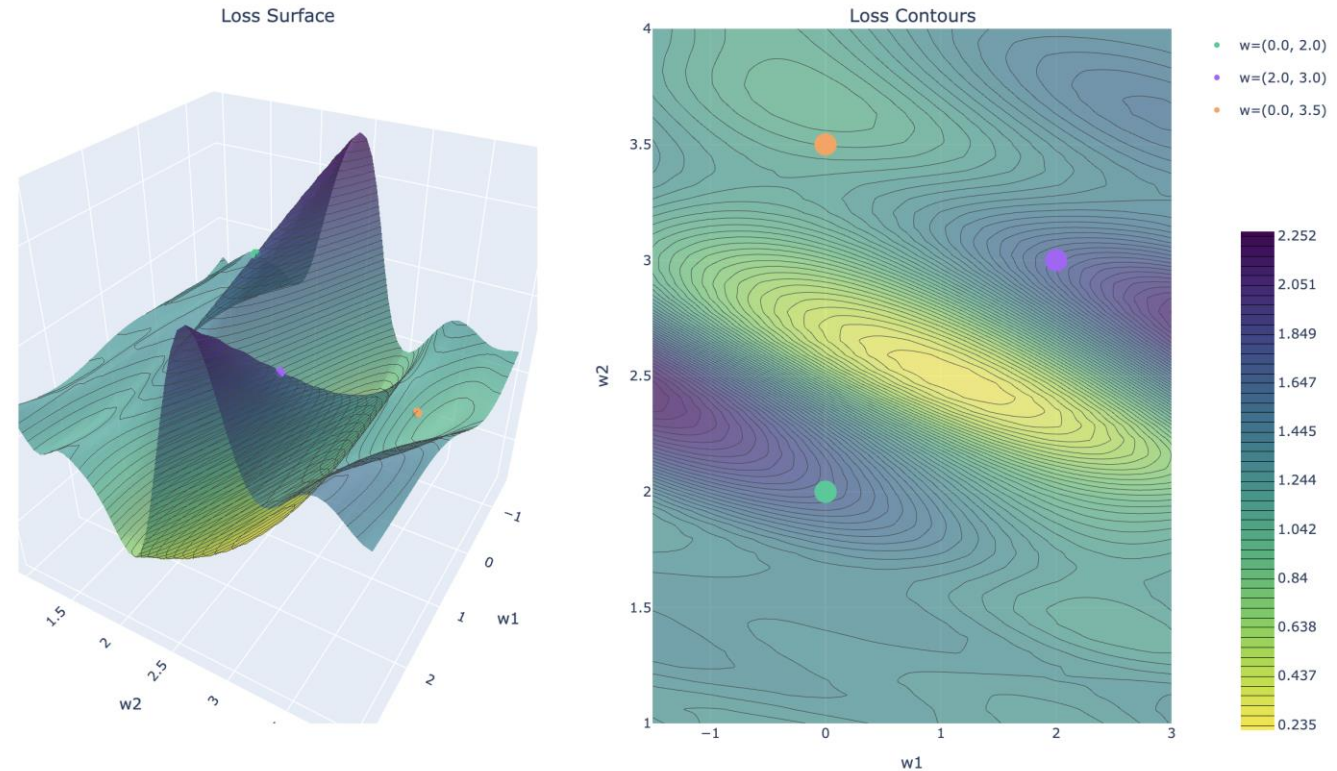


1809914

The neural network loss surface is **highly non-convex** so choice of starting point can result in:

- diverging solutions
- slow convergence
- bad local minima

What makes a good starting point?



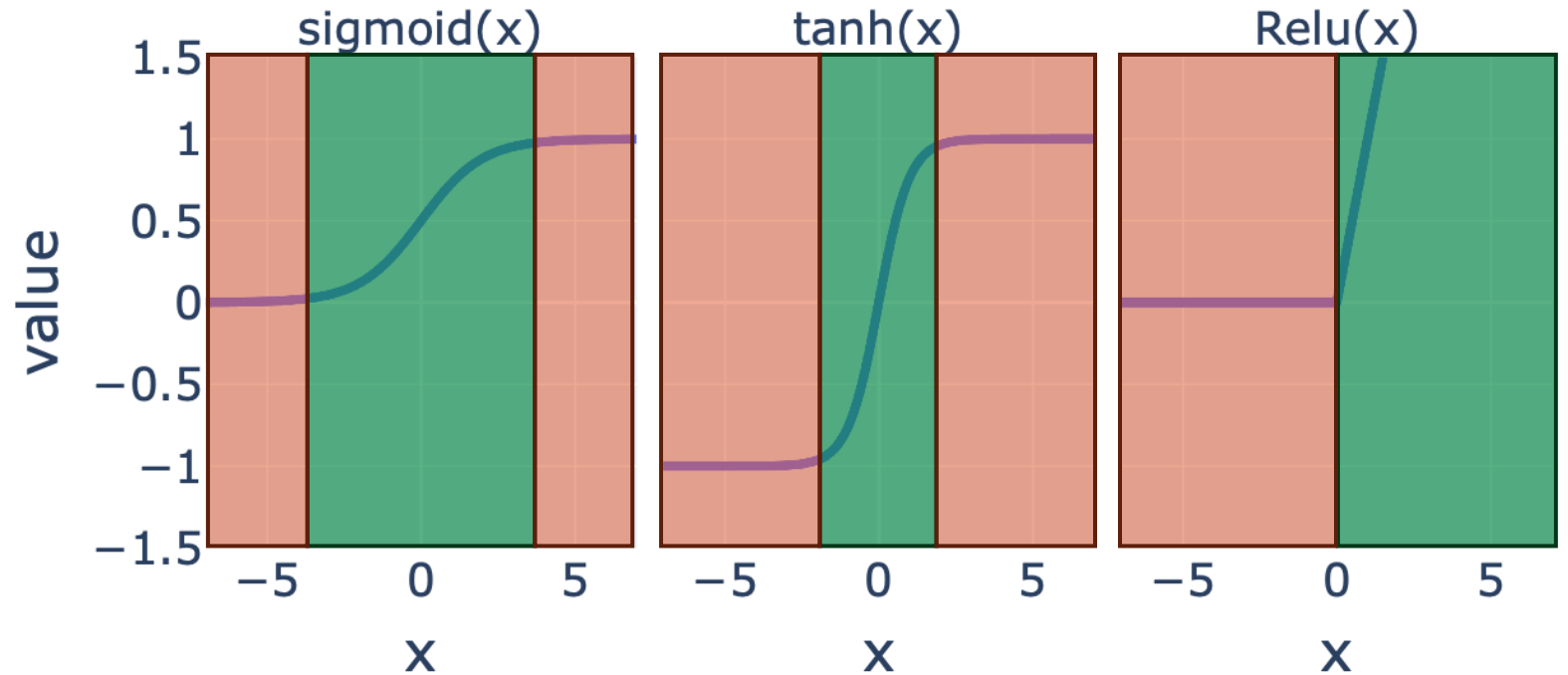


1809914

“Sweet-spot” of the Non-Linearity

Sweet-spot is near the **transition point** in the non-linear transformation.

- Non-zero gradient
- Typically, **near zero pre-activations**



Outside the transition point the activation is **saturated**.

- Small changes don't change activation.

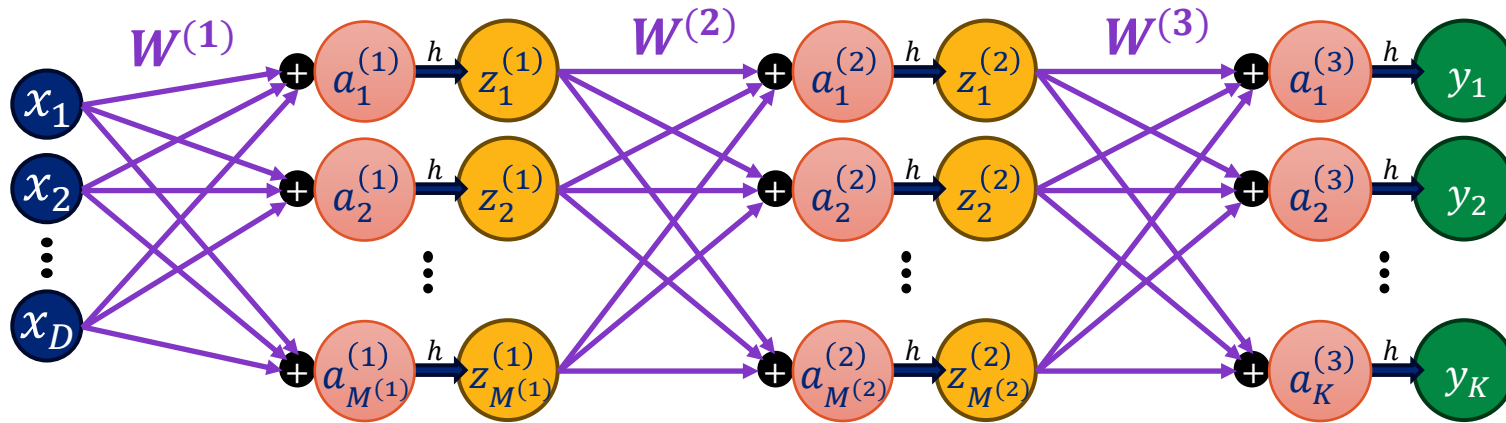


1809914

Starting at Zero-Weights

If we initialize all weights to zero, the pre-activations will all be zero (at the **transition point**).

Example: Two hidden-layer neural network



$$a^{(1)} = W^{(1)}x \quad z^{(1)} = h(a^{(1)})$$

$$a^{(2)} = W^{(2)}z^{(1)} \quad z^{(2)} = h(a^{(2)})$$

$$a^{(3)} = W^{(3)}z^{(2)} \quad y = h(a^{(3)})$$

$$y = h\left(W^{(3)}h\left(W^{(2)}h\left(W^{(1)}x\right)\right)\right)$$

- If all $W^{(l)} = 0$ then all the **pre-activations** $a^{(l)} = 0$ are also zero.



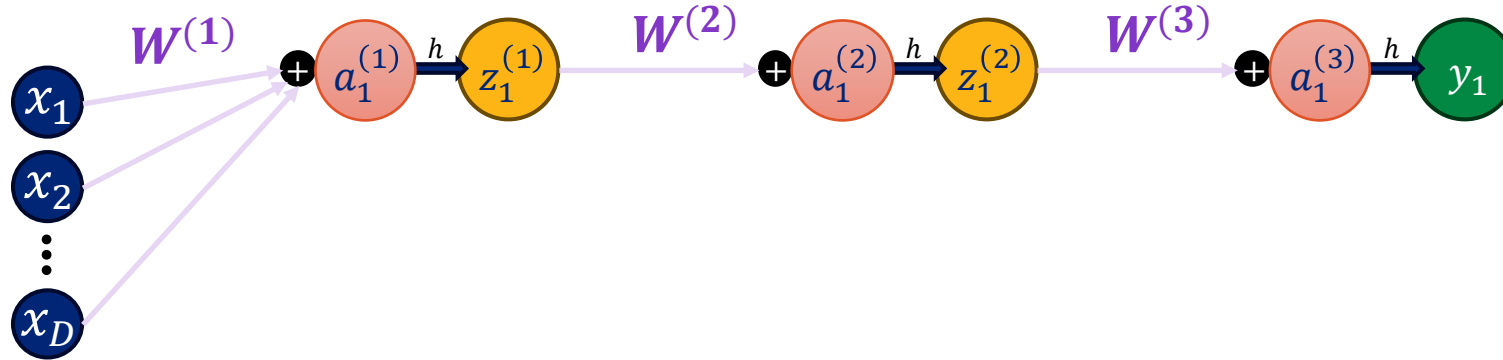
Why is initializing all neural network weights to zero generally a bad idea?

Unbroken Symmetry at Zero-Weights



1809914

If we initialize all weights to zero, neurons within each hidden layer **produce identical activations** and **remain identical during training**.



If we use a **relu** or **tanh** activation the gradient will be zero.

$$\frac{\partial L(t, y(x; w))}{\partial W_{km}^{(3)}} = L'(t, y(x; w)) \left(\frac{\partial y(x; w)}{\partial W_{km}^{(3)}} \right) = L'(t, y(x; w)) h'(a^{(3)}) h(a^{(2)})_{km}^0$$

- **No learning of input-dependent features**

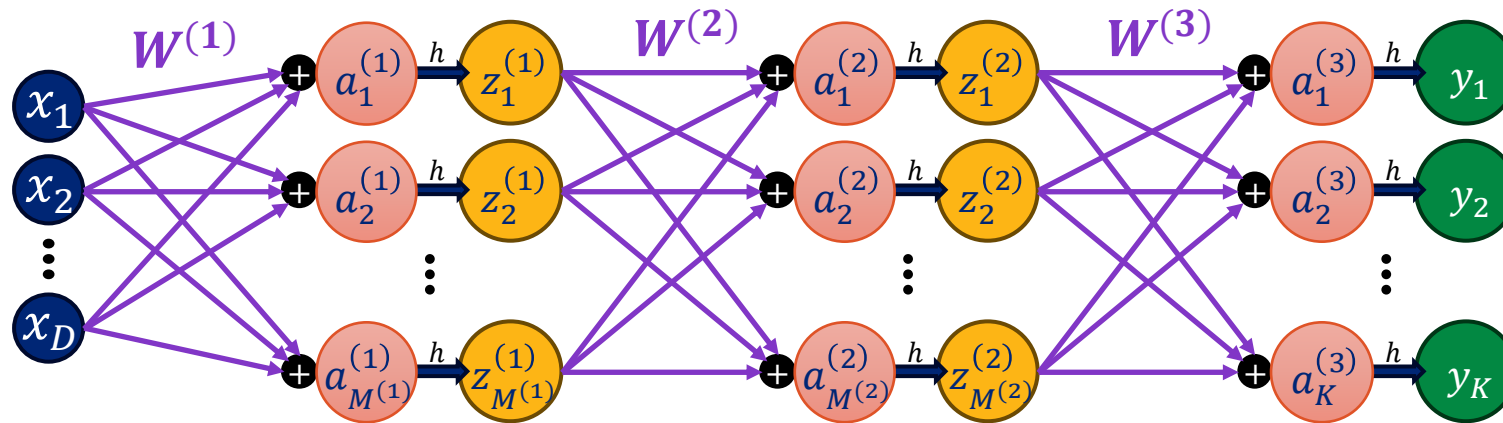
Randomization is needed to **break symmetry** across neurons.

Weight Symmetry In Neural Networks



1809914

For any layer in the neural network, we could “re-arrange” the neurons to obtain an identical network.

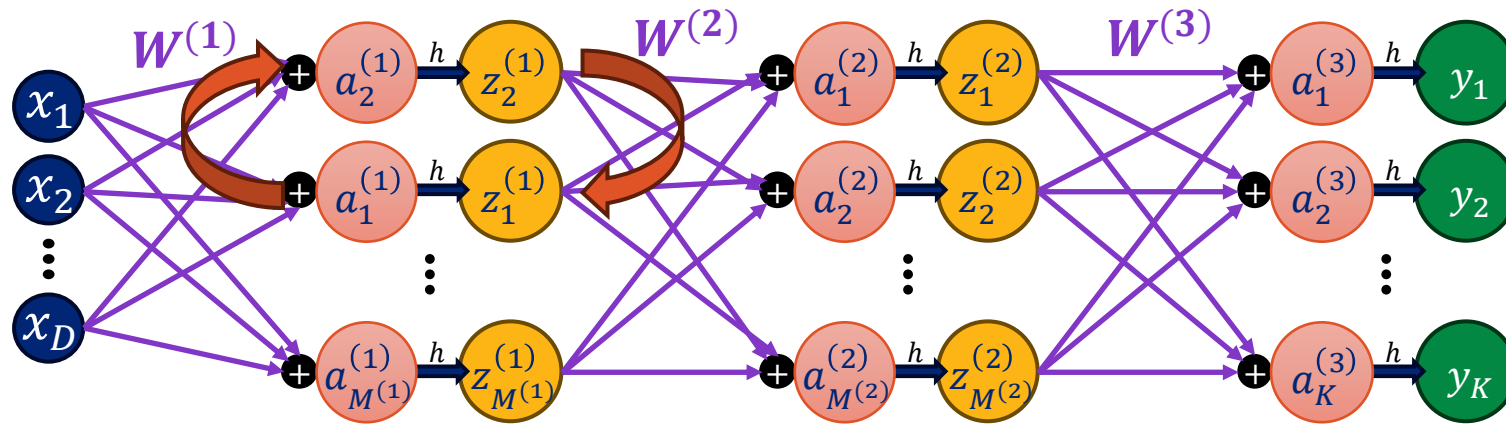


Weight Symmetry In Neural Networks

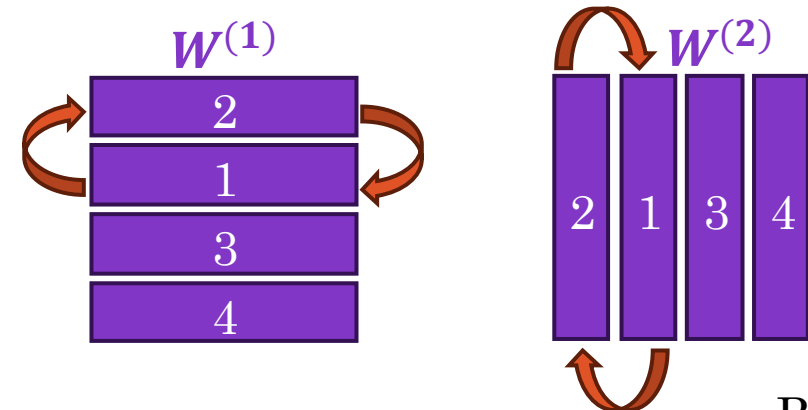


1809914

For any layer in the neural network, we could “re-arrange” the neurons to obtain an identical network.



- This corresponds to **permuting rows and columns** of the weight matrices
- For each layer l we can permute the $M^{(l)}$ **neurons** $M^{(l)}!$ **ways**.

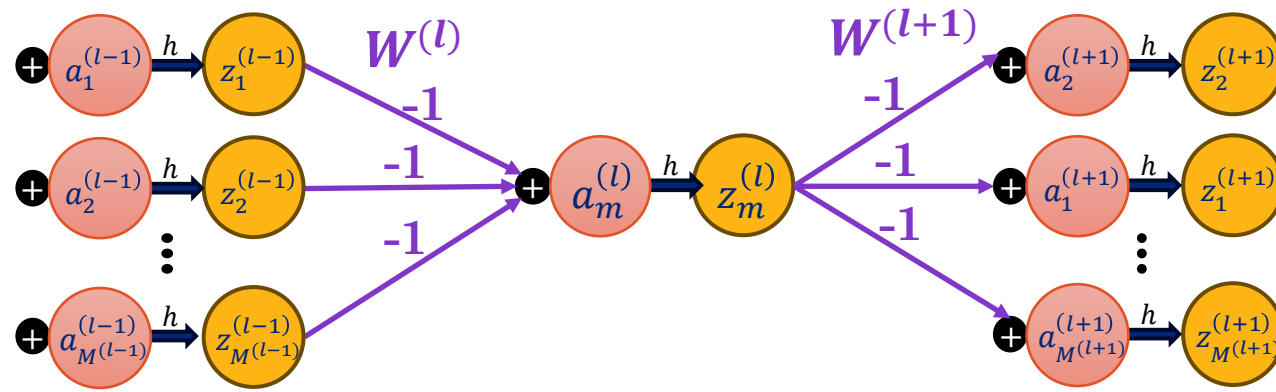




1809914

Sign Flipping Equivalence in Networks

For **odd activation functions** where $h(-a) = -h(a)$ (e.g., tanh), we can **negate the weights** to produce equivalent networks:



$$\begin{aligned} a^{(l+1)} &= W^{(l+1)} h(W^{(l)} z^{(l-1)}) \\ &= (-W^{(l+1)}) h((-W^{(l)}) z^{(l-1)}) \end{aligned}$$

- This can be done for individual neurons corresponding to rows of $-W^{(l)}$ and columns of $-W^{(l+1)}$
- Sign flipping produces an additional $2^{M^{(l)}}$ **equivalent networks** for layer l .



1809914

Counting Network Symmetries

Summarizing the two primary symmetries for each layer l with $M^{(l)}$ **neurons**:

- **Permutations:** $M^{(l)}!$
- **Sign Flipping:** $2^{M^{(l)}}$

Therefore, a **single parameterization** has MANY equivalent networks:

$$\prod_{l=1}^L (M^{(l)}!) (2^{M^{(l)}})$$

Is this an issue?

- No, we just need to find a “good parameterization”
- The **good parameterizations** have **many equivalent forms**



1809914

Breaking Symmetry with Random Weight Initialization

The primary goal of weight initialization is to **break weight symmetry**

Weights are **randomly initialized** by sampling from a distribution:

- Uniform $[-\epsilon, \epsilon]$, which has variance $\epsilon^2/3$
- $\mathcal{N}(0, \epsilon^2)$, which has variance ϵ^2

Biases are typically initialized at zero.

- No need to break symmetry for the bias term.

The **choice of ϵ is important** as we want to ensure that the activations (and gradients) don't become too large (**explode**) or too small (**vanish**) across layers.

- **Mathematically:** keep the **variance constant** across layers.

Variance of the Pre-activations

Pre-activations $a_m^{(l)}$ and activations $z_m^{(l)}$ on layer l :

$$a_m^{(l)} = \sum_{i=1}^{M^{(l-1)}} W_{mi}^{(l)} z_i^{(l-1)}; \quad z_m^{(l)} = \text{ReLU}(a_m^{(l)})$$

- **Assume:** the $W_{mi}^{(l)}$ are IID with $\mathbb{E}[W_{mi}^{(l)}] = 0$ and $\text{Var}[W_{mi}^{(l)}] = \mathbb{E}\left[\left(W_{mi}^{(l)}\right)^2\right] = \epsilon^2$ and that $W_{mi}^{(l)}$ are independent of the inputs $z_i^{(l-1)}$ and $z_i^{(l-1)}$ are IID.

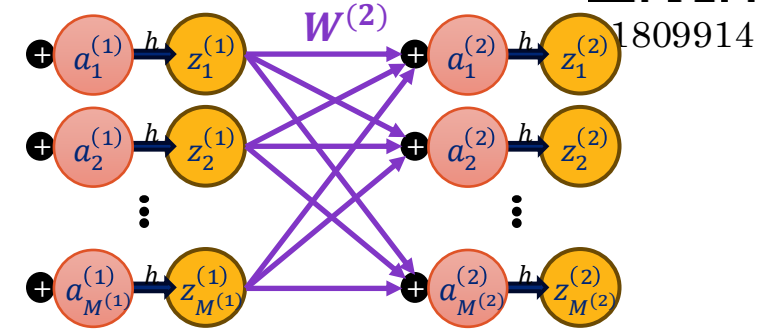
Because the terms of the sum are **independent**, the variance of the sum is:

$$\text{Var}[a_m^{(l)}] = \sum_{i=1}^{M^{(l-1)}} \text{Var}[W_{mi}^{(l)} z_i^{(l-1)}]$$

Using $\text{Var}[A] = \mathbb{E}[A^2] - \mathbb{E}[A]^2$, the independence of W and z :

$$\text{Var}[Wz] = \mathbb{E}[W^2] \mathbb{E}[z^2] - \mathbb{E}[W]^2 \mathbb{E}[z]^2 = \text{Var}[W] \mathbb{E}[z^2] = \epsilon^2 \mathbb{E}[z^2]$$

Summing the $M^{(l-1)}$ identical terms: $\text{Var}[a_m^{(l)}] = M^{(l-1)} \epsilon^2 \mathbb{E}\left[\left(z_i^{(l-1)}\right)^2\right]$





1809914

He Initialization for ReLU Activations

Pre-activations $a_m^{(l)}$ and activations $z_m^{(l)}$ on layer l , and we just derived $\text{Var}[a_m^{(l)}]$

$$a_m^{(l)} = \sum_{i=1}^{M^{(l-1)}} W_{mi}^{(l)} z_i^{(l-1)}; \quad z_m^{(l)} = \text{ReLU}(a_m^{(l)}); \quad \text{Var}[a_m^{(l)}] = M^{(l-1)} \epsilon^2 \mathbb{E}[(z_i^{(l-1)})^2]$$

We now need to compute $\mathbb{E}[(z_i^{(l-1)})^2] = \mathbb{E}[\text{ReLU}(a_i^{(l-1)})^2]$. If we let $p(a)$ be the density of $a = a_i^{(l-1)}$, symmetric about zero, $p(-a) = p(a)$, with variance λ^2 . Since $\text{ReLU}(a) = 0$ for $a < 0$:

$$\mathbb{E}[z^2] = \int_{-\infty}^{\infty} \text{ReLU}(a)^2 p(a) da = \int_0^{\infty} a^2 p(a) da = \frac{1}{2} \int_{-\infty}^{\infty} a^2 p(a) da = \frac{1}{2} \lambda^2$$

Therefore, $\text{Var}[a_m^{(l)}] = \frac{M^{(l-1)}}{2} \epsilon^2 \lambda^2$ and to preserve the variance across layers:

$$\frac{M^{(l-1)}}{2} \epsilon^2 \lambda^2 = \lambda^2 \Rightarrow \epsilon^2 = \frac{2}{M^{(l-1)}}$$

He initialization is then $W_{mi}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{M^{(l-1)}}\right)$ or $W_{mi}^{(l)} \sim \text{Uniform}\left[-\sqrt{\frac{6}{M^{(l-1)}}}, \sqrt{\frac{6}{M^{(l-1)}}}\right]$

Xavier Initialization for Tanh Activations



1809914

Preserving the variance forward needs $\epsilon^2 = 1/M^{(l-1)}$, backward $\epsilon^2 = 1/M^{(l)}$. For **tanh**, Glorot and Bengio (2010) compromise:

$$\epsilon^2 = \frac{2}{M^{(l)} + M^{(l-1)}}$$

Fan-out Fan-in

Xavier initialization is accomplished by sampling weights according to:

$$W_{mi}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{M^{(l)} + M^{(l-1)}}\right)$$

$$W_{mi}^{(l)} \sim \text{Uniform}\left[-\sqrt{\frac{6}{M^{(l)} + M^{(l-1)}}}, \sqrt{\frac{6}{M^{(l)} + M^{(l-1)}}}\right]$$



1809914

Comparison Between Xavier and He

He Initialization:

Compensates for ReLU activation function

$$\epsilon^2 = \frac{2}{M^{(l-1)}}$$

Xavier Initialization:

Balances forward and backward variance for approx. linear activations

$$\epsilon^2 = \frac{2}{M^{(l)} + M^{(l-1)}}$$

Other factors to consider:

- **Output layer:** often initialized with **smaller** (or zero) weights, so the first predictions are **uncertain** (logits near 0).
- **Normalization layers** make training **less sensitive** to the initialization.
- **Residual connections** (Lecture 15): the variance **grows with depth**, so often scale the variance by $1/L$ for L residual layers.

Weight Initialization in PyTorch



```
class MLPModel(nn.Module):
    def __init__(self, dims, activation="relu"):
        super().__init__()
        self.activation = activation
        self.layers = nn.ModuleList(
            [nn.Linear(dims[i], dims[i + 1]) for i in range(len(dims) - 1)])
        self.act = torch.relu if activation == "relu" else torch.tanh
        self.reset_parameters()

    def reset_parameters(self):
        for layer in self.layers:
            if self.activation == "relu":
                nn.init.kaiming_normal_(layer.weight, nonlinearity="relu") # He
            else:
                nn.init.xavier_normal_(layer.weight) # Xavier
            nn.init.zeros_(layer.bias)

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i < len(self.layers) - 1:
                x = self.act(x)
        return x
```

Roadmap

- PyTorch
- Parameter Initialization
- Normalization



1809914



1809914

Data Normalization

Earlier we saw that inputs to a network (or layer) that vary significantly in scale can lead to slow convergence.

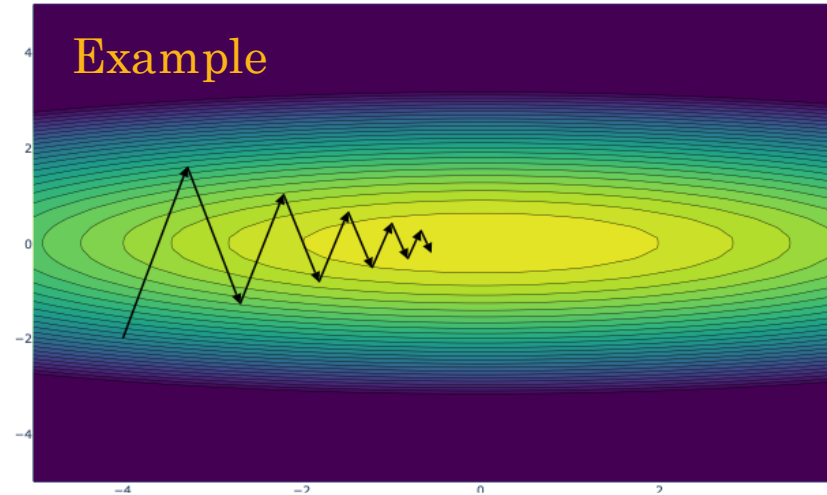
One solution is **feature standardization**:

$$\hat{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i}$$

where μ_i and σ_i are computed from the training data:

$$\mu_i = \frac{1}{N} \sum_{n=1}^N x_{ni} \quad \sigma_i^2 = \frac{1}{N} \sum_{n=1}^N (x_{ni} - \mu_i)^2$$

Then, at test-time we use the same μ_i and σ_i .





1809914

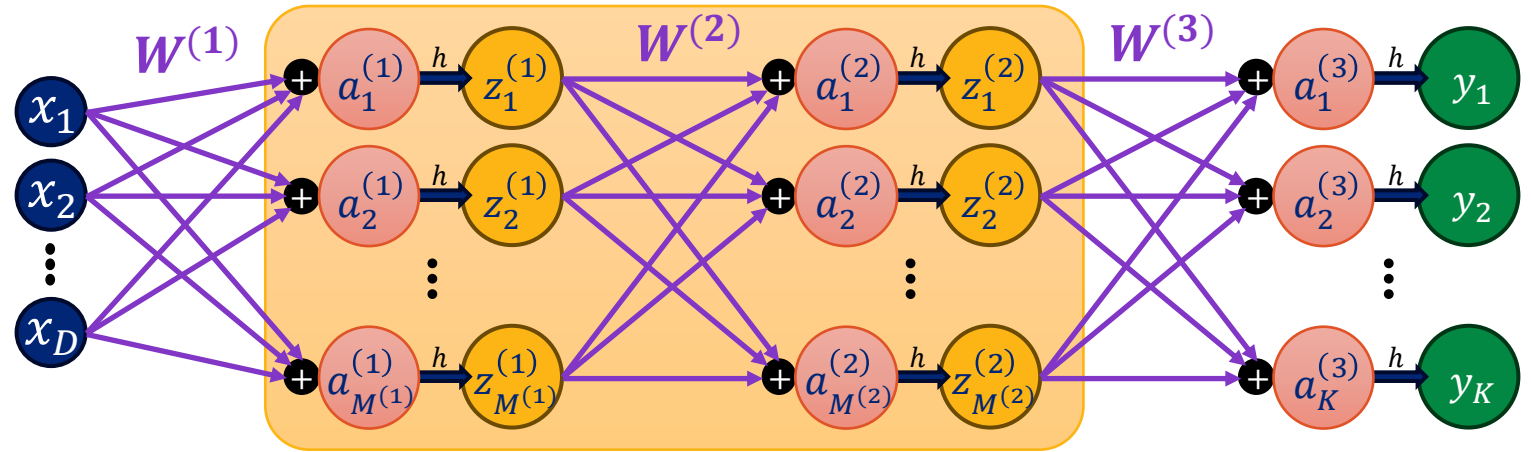
Normalization in the Network

Could we apply normalization to interior layers of the network?

Example:

- 2-hidden layer NN

$$y = h\left(W^{(3)}h\left(W^{(2)}h\left(W^{(1)}x\right)\right)\right)$$



Why would **hidden layer normalization** help?

- Ensure internal activations are similar scale
 - Chain rule $\rightarrow$ gradients stay similar scale

Problem: internal activations change during learning (SGD)!



1809914

Batch Normalization

Apply **normalization** (standardization) to **activations** $z_{km}^{(l)}$ during learning by computing $\mu_m^{(l)}$ and $\sigma_m^{(l)}$ within a **mini-batch** of SGD.

$$\hat{z}_{km}^{(l)} = \frac{z_{km}^{(l)} - \mu_m^{(l)}}{\sqrt{(\sigma_m^{(l)})^2 + \delta}}$$

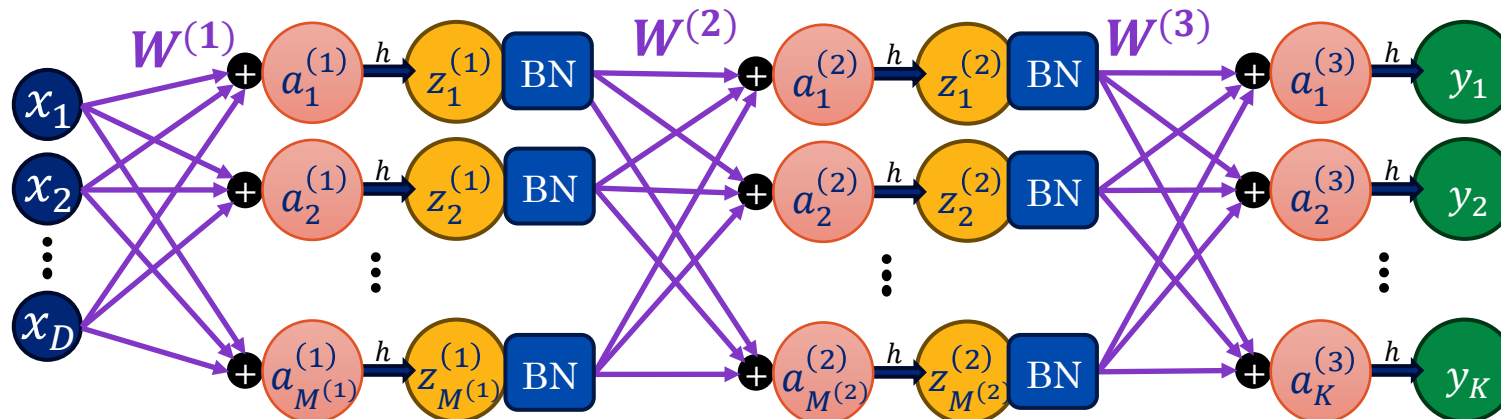
Activation m on layer l for the k^{th} datapoint in the mini-batch

$$\mu_m^{(l)} = \frac{1}{K} \sum_{k=1}^K z_{km}^{(l)}$$

Mean over the mini-batch of K datapoints

$$(\sigma_m^{(l)})^2 = \frac{1}{K} \sum_{k=1}^K (z_{km}^{(l)} - \mu_m^{(l)})^2$$

Variance over the mini-batch of K datapoints





1809914

Making Predictions with Batch Norm.

To make a prediction at test time, we need the mean $\mu_m^{(l)}$ and variance $\left(\sigma_m^{(l)}\right)^2$ of the hidden layer activations from training.

Ideally, we would compute them from the entire training dataset.

- Expensive

Solution: compute running exponential averages during training:

$$\mu_m^{(l,\tau)} = \alpha \left(\mu_m^{(l,\tau-1)} \right) + (1 - \alpha) \frac{1}{K} \sum_{k=1}^K z_{km}^{(l)}$$

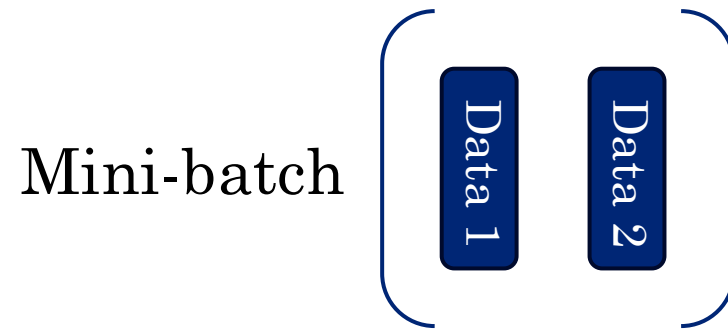
$$\left(\sigma_m^{(l,\tau)} \right)^2 = \alpha \left(\sigma_m^{(l,\tau-1)} \right)^2 + (1 - \alpha) \frac{1}{K} \sum_{k=1}^K \left(z_{km}^{(l)} - \mu_m^{(l)} \right)^2$$

Batch Normalization is Difficult to Parallelize



1809914

Data-parallel stochastic gradient descent

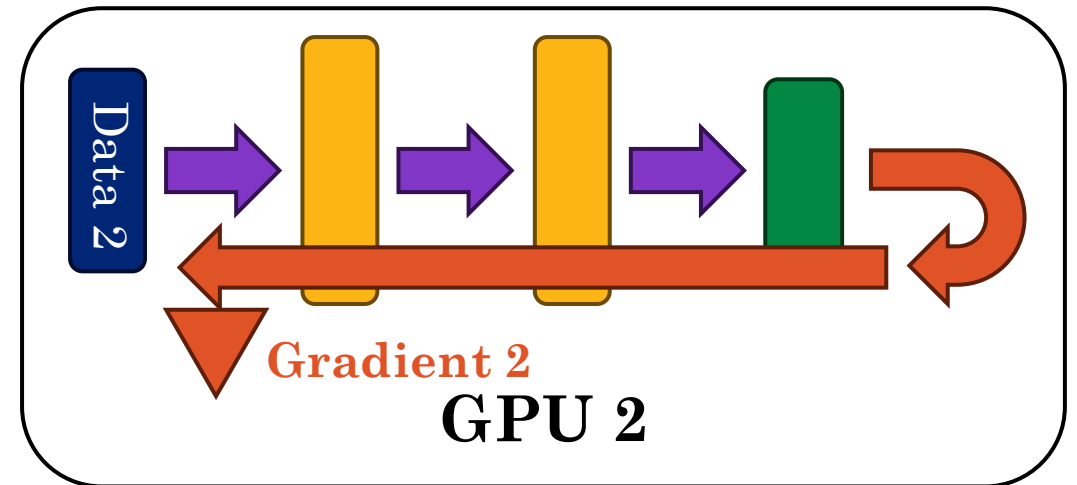
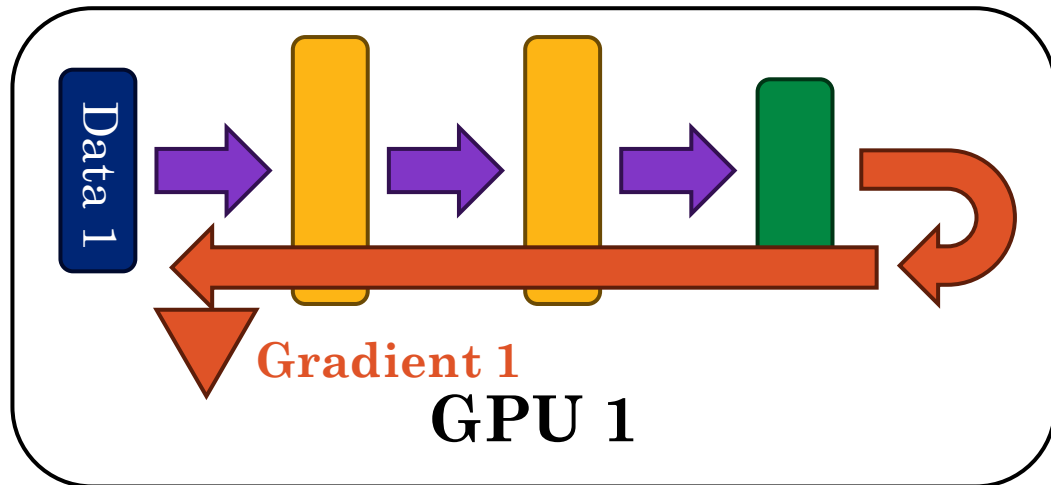


Batch Normalization is Difficult to Parallelize



1809914

Data-parallel stochastic gradient descent

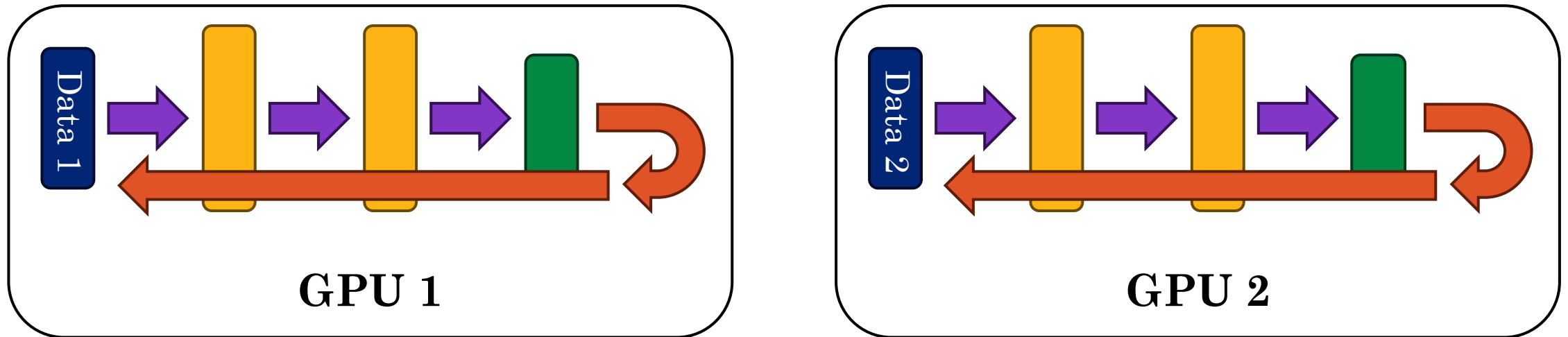


Batch Normalization is Difficult to Parallelize



1809914

Data-parallel stochastic gradient descent



GPUs share gradients and compute sum (called **all-reduce**).

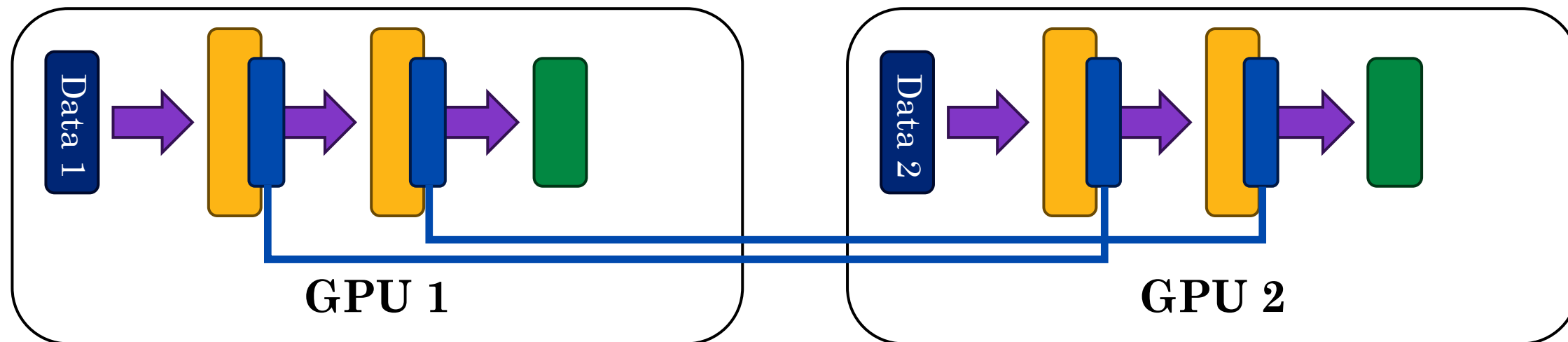
$$\nabla \text{Gradient 1} + \nabla \text{Gradient 2}$$

Batch Normalization is Difficult to Parallelize



1809914

Batch normalization **couples the data points** of a mini-batch.



Cannot compute gradients **independently** across GPUs



1809914

Layer Normalization

Layer Normalization computes the layer mean and variance across the **neurons in the layer** instead of across the mini-batch:

$$\hat{z}_{km}^{(l)} = \frac{z_{km}^{(l)} - \mu_k^{(l)}}{\sqrt{(\sigma_k^{(l)})^2 + \delta}}$$

Activation m on layer l
for the k^{th} datapoint in
the mini-batch

$$\mu_k^{(l)} = \frac{1}{M^{(l)}} \sum_{m=1}^{M^{(l)}} z_{km}^{(l)}$$

Mean over the $M^{(l)}$
hidden activations

$$(\sigma_k^{(l)})^2 = \frac{1}{M^{(l)}} \sum_{m=1}^{M^{(l)}} (z_{km}^{(l)} - \mu_k^{(l)})^2$$

Variance over the $M^{(l)}$
hidden activations

Normalization is applied the same way at test-time.

- No need to estimate on training data.

Layer norm (or RMSNorm) is used in **transformers** and **LLMs**.



1809914

RMS Normalization (RMSNorm)

Layer norm re-centers (subtracts $\mu_k^{(l)}$) and re-scales (divides by $\sigma_k^{(l)}$). Zhang and Sennrich (2019) found that the **re-scaling** provided most of the benefit.

RMSNorm skips the mean and divides by the **root mean square** of the activations:

$$\text{RMS}_k^{(l)} = \sqrt{\frac{1}{M^{(l)}} \sum_{m=1}^{M^{(l)}} \left(z_{km}^{(l)}\right)^2 + \delta} \quad \tilde{z}_{km}^{(l)} = \gamma_m^{(l)} \frac{z_{km}^{(l)}}{\text{RMS}_k^{(l)}}$$

Invariant to re-scaling $z_{km}^{(l)} \rightarrow c z_{km}^{(l)}$ ($c > 0$, up to δ), but **not to shifts**.

- Equals layer norm without β when $\mu_k^{(l)} = 0$.

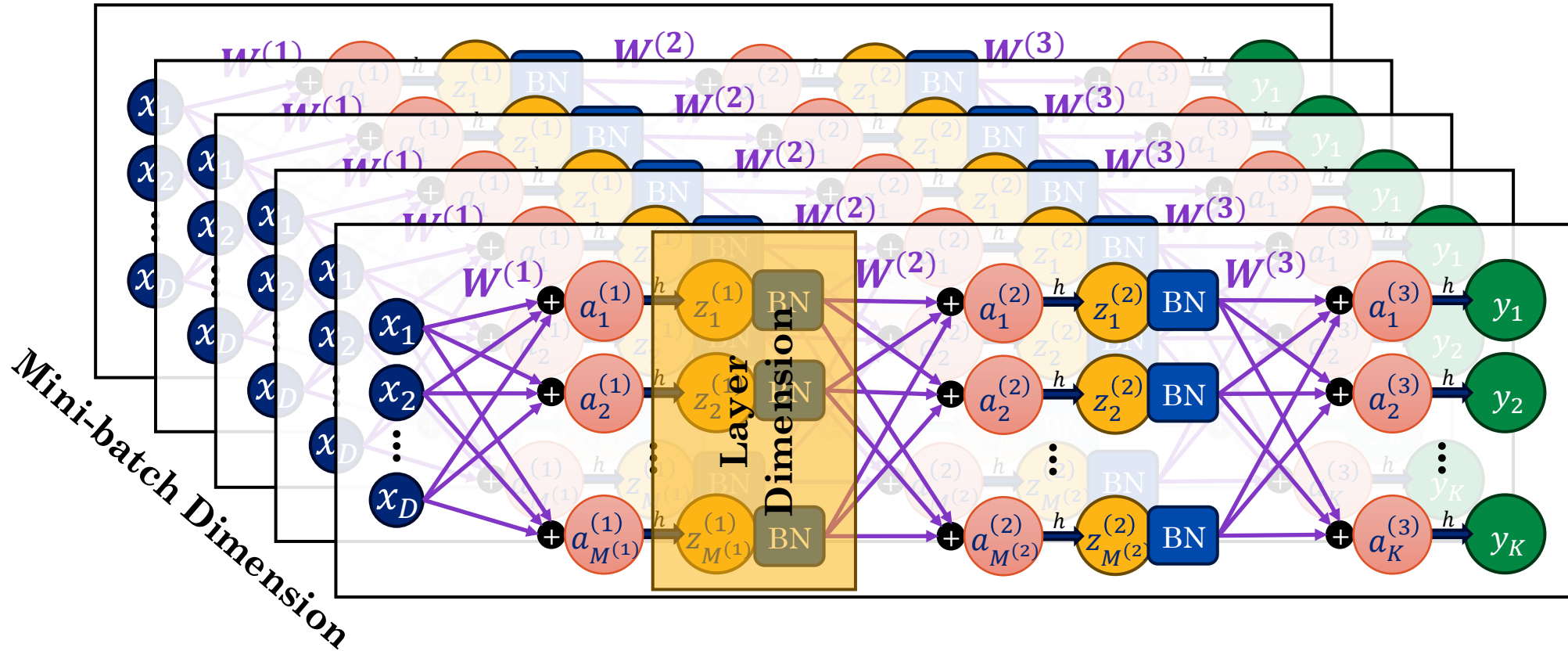
Used in most recent LLMs (e.g., Llama, Mistral, Gemma).

```
norm = nn.RMSNorm(c) # learned gamma, no beta
```

Layer Normalization vs Batch Normalization



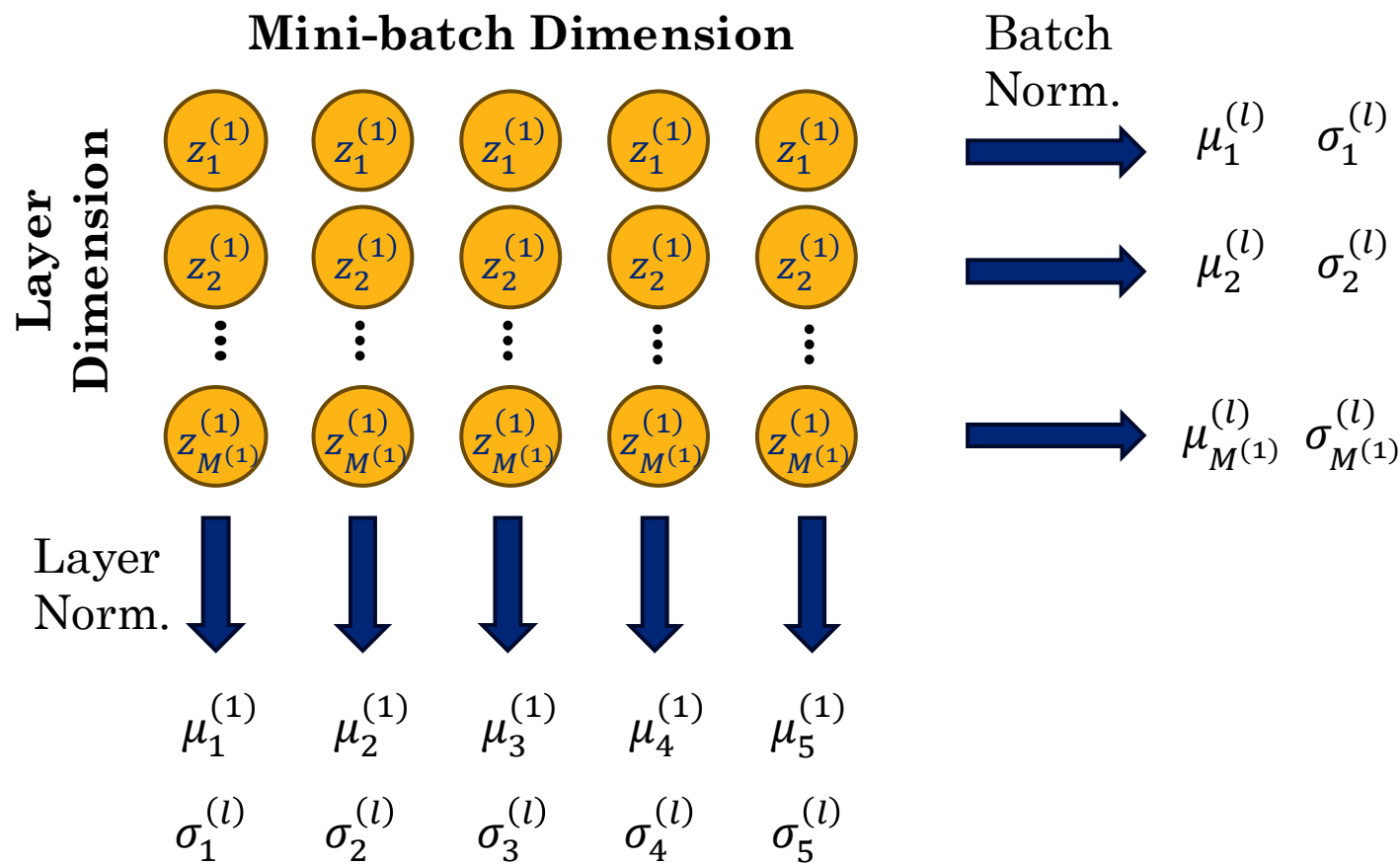
1809914



Layer Normalization vs Batch Normalization



1809914





1809914

Re-Scaling with Normalization

Normalization limits the **degrees of freedom** of a layer.

- For example, by centering the distribution it is no longer possible for all activations to be positive.

Solution: Learned scale $\gamma_m^{(l)}$ and shift $\beta_m^{(l)}$ parameters

$$\tilde{z}_{km}^{(l)} = \gamma_m^{(l)} \hat{z}_{km}^{(l)} + \beta_m^{(l)}$$

The parameters $\gamma_m^{(l)}$ and $\beta_m^{(l)}$ are learned through SGD.

- This generally improves convergence.

Batch-Norm and Layer-Norm in PyTorch



1809914

Creating norm layers:

```
if norm_kind == "batch":
    self.norms = nn.ModuleList([nn.BatchNorm1d(c) for c in dims[1:-1]])
else: # norm_kind == "layer"
    self.norms = nn.ModuleList([nn.LayerNorm(c) for c in dims[1:-1]])
```

Using norm layers:

```
def forward(self, x):
    for i, layer in enumerate(self.layers):
        x = layer(x)
        if i < len(self.layers) - 1:
            x = self.norms[i](x)
            x = self.act(x)
    return x
```



1809914

Demo

Notebook, Part 3: batch and layer normalization by hand and in PyTorch.

Lecture 14

PyTorch, Initialization, and Normalization

Reading: Bishop, Sections 6.2.4, 7.2.5, and 7.4. PyTorch basics:
docs.pytorch.org/tutorials/beginner/basics