Lecture 3: Training Deep Networks in Practice

Agenda

Focus on practical techniques and considerations for training, both in terms of optimization and generalization

- Weight Decay
- Dropout
- Choice of activations
- Parameter Initialization
- Batchnorm
- WeightNorm
- Data Augmentation

Weight Decay and L2

Standard SGD update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{w_t} \hat{\mathcal{L}} - \alpha * \lambda \mathbf{w}_t$$

- Weight decay for vanilla SGD corresponds to an L2 regularization
- Important parameter in many models
- It’s exact effect on training has been shown to be more complex than just regularization

Weight Decay

- Krizhevsky et al (AlexNet) observed that it improves training accuracy not just testing accuracy

- Weight decay is included in the optimizer abstraction in torch and other frameworks

- Not equivalent to L2 outside of vanilla SGD

Zhang, Guodong, et al. "Three mechanisms of weight decay regularization." *ICLR* 2018).

Weight Decay and Adaptive Optimizers

- Although in standard SGD it is equivalent to L2 regularization with momentum or adaptive optimizers it is no longer equivalent

SGD + Momentum

$$\min \mathcal{L}(X, Y, w)$$

$$\mathbf{g} = \nabla_w \mathcal{L}(X, Y, w_t)$$

$$\mathbf{v}_{t+1} = \mu * \mathbf{v}_t + (1 - \mu) * \mathbf{g}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha * \mathbf{v}_{t+1}$$

$$\min \mathcal{L}(X, Y, w) + \frac{\lambda}{2}\|w\|^2$$

$$\mathbf{g} = \nabla_w \mathcal{L}(X, Y, w_t) + \lambda w_t$$

$$\mathbf{v}_{t+1} = \mu * \mathbf{v}_t + (1 - \mu)\mathbf{g}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha * \mathbf{v}_{t+1}$$

SGD + Momentum + Weight Decay

$$\min \mathcal{L}(X, Y, w)$$

$$\mathbf{g} = \nabla_w \mathcal{L}(X, Y, w_t)$$

$$\mathbf{v}_{t+1} = \mu * \mathbf{v}_t + (1 - \mu)\mathbf{g}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha * \mathbf{v}_{t+1} - \alpha\lambda\mathbf{w}_t$$

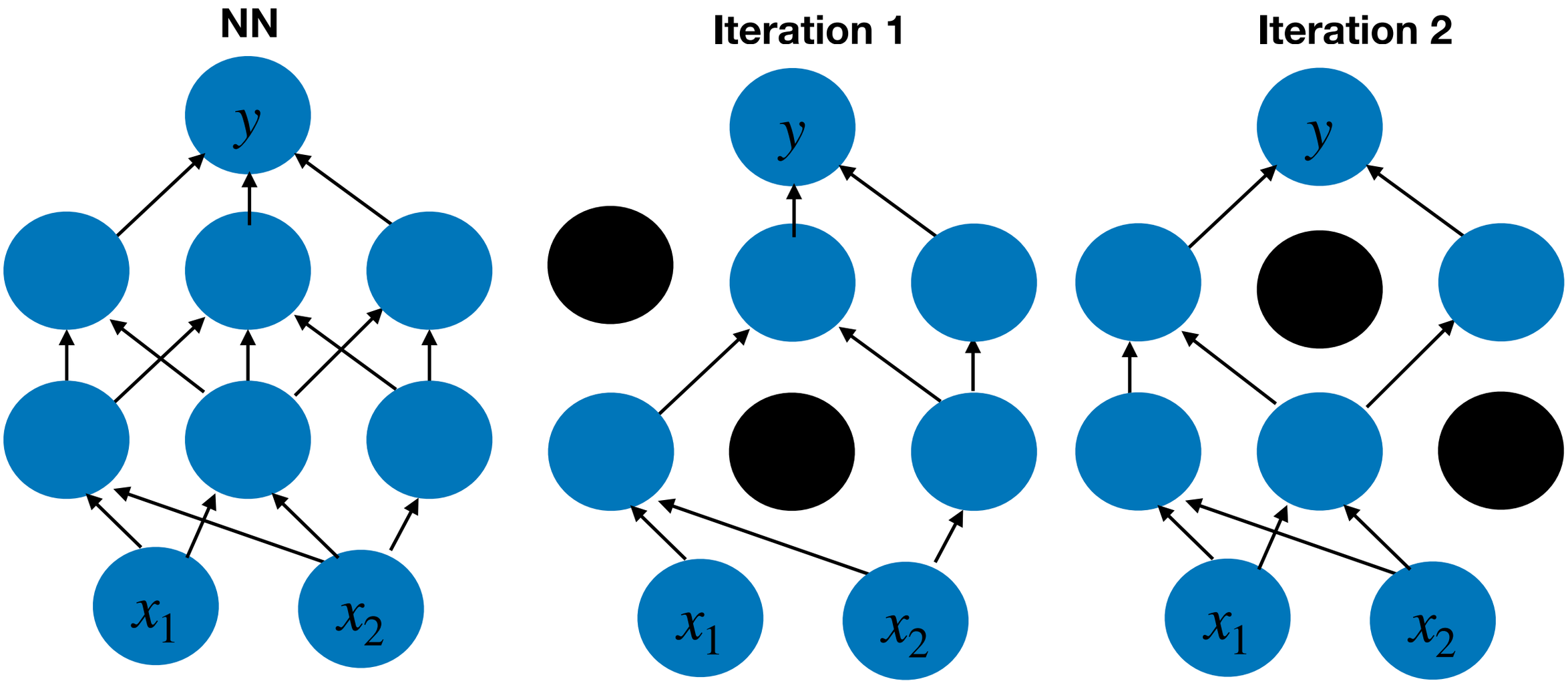
Weight Decay and Adaptive Optimizers

- Observed that weight decay is more effective for regularization than L2 in non-vanilla SGD

- The exact mechanisms leading to its improvement are not well understood, some hypothesis
  - Leads to greater effective learning rate
  - Regularizing the network input-output Jacobean

Zhang, Guodong, et al. "Three mechanisms of weight decay regularization." *ICLR* 2018).

Dropout



- zero out some outputs et each layer with probability (1-p)

Dropout Interpretation

- Can be seen as introducing noise to the training process to increase robustness

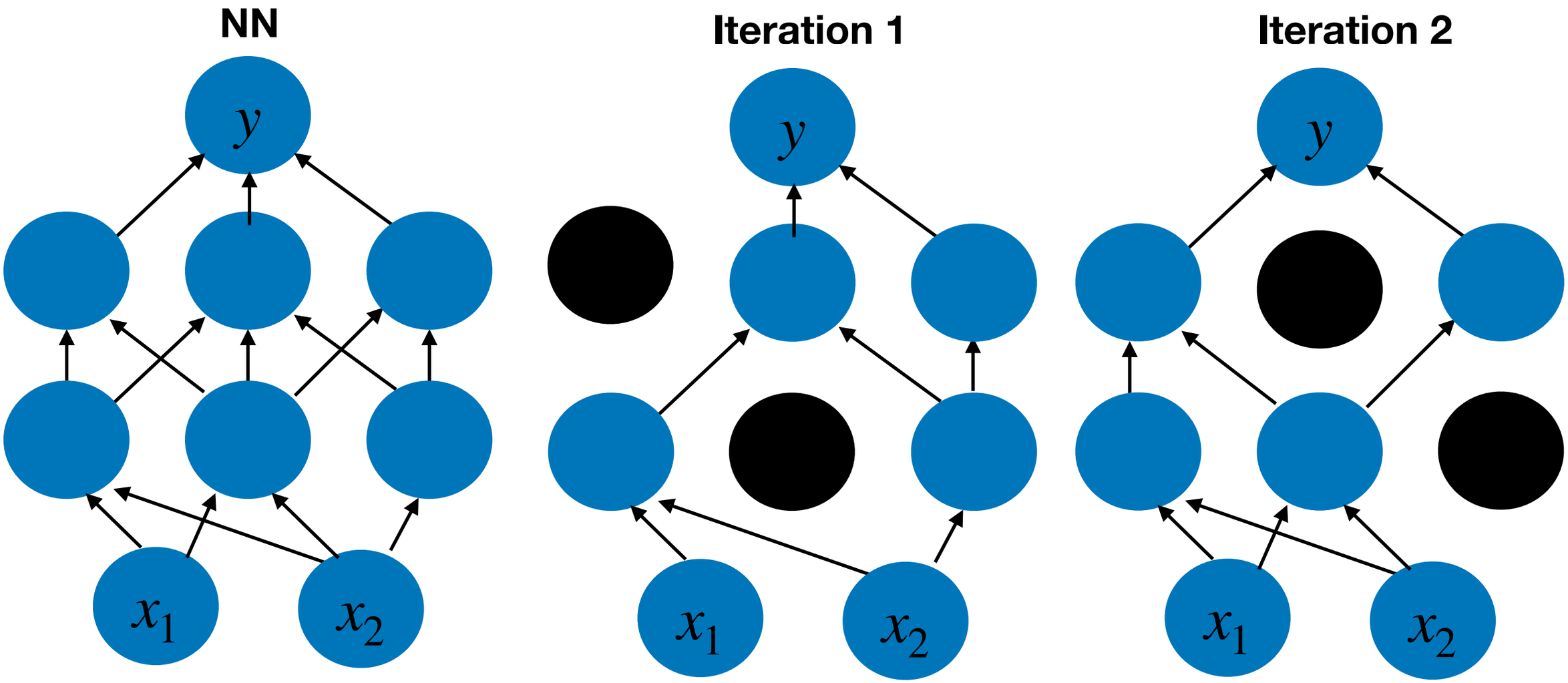
- Avoid relying on a single weight overfitting

- Another common interpretation of Dropout is creating an ensemble of models with overlapping parameters

  - This interpretation has lead to applications outside of regularization

  - Test-time Dropout for uncertainty estimation Gal et al

Dropout Inference



- At inference time we use all hidden nodes but rescale the output by p

Different Train / Test Behaviour

```
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.linear1 = nn.Linear(784,200)
        self.dropout = nn.Dropout(0.5)
        self.linear2 = nn.Linear(200,10)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.dropout(x)
        return self.linear2(x)

loss = nn.CrossEntropyLoss()
model = NN()
```

```
dropout = nn.Dropout(0.5)
dropout.train()
dropout.training
```

```
True
```

```
dropout.eval()
dropout.training
```

```
False
```

Different Train / Test Behaviour

```
model.train()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(train_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()

model.eval()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(test_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

- Any nn modules within the model which define a different train test behaviour will set training=False/True based on .eval() and .train()

Dropout

- Dropout is less commonly used these days in large scale problems
- Still a useful technique in some cases, particularly in small datasets

Initialization

- In deep learning the parameter initialization can be critical
  - Fight vanishing gradient problem
  - Ill-conditioning
- Poor initialization can prevent learning from proceeding at all
- Poor initialization can cause learning to be slow

Example of poor initialization

$$y = \mathbf{w}_2^T \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}) \qquad \mathbf{W}_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \mathbf{w_2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial y}{\partial \mathbf{W}_1} = \mathbf{w}_2 \circ (1 - tanh^2(\mathbf{W}_1\mathbf{x} + b))\mathbf{x}^T$$

$$\mathbf{x} = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix}$$

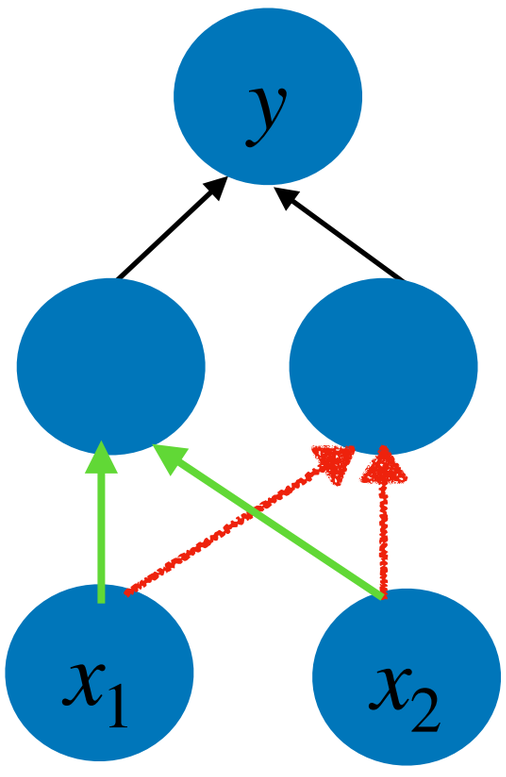
$$\frac{\partial y}{\partial \mathbf{W}_1} = \mathbf{w}_2 \circ (1 - tanh^2(0))\mathbf{x}^T = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Example of poor initialization

$$y = \mathbf{w}_2^T \rho(\mathbf{W}_1 \mathbf{x} + \mathbf{b}) \qquad \mathbf{W}_1 = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix} \quad \mathbf{w}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

What happens if a1=b1,a2=b2

$$\frac{\partial y}{\partial \mathbf{W}_1} = \mathbf{w}_2 \circ (\rho'(\mathbf{W}_1 \mathbf{x} + b))\mathbf{x}^T$$



Good Init can help fight Vanishing Gradient

- Common initialization is based on layer size
  - Xavier initialization $W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$
  - Kaiming initialization $W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$ **Designed for Relu**
- We typically initialize biases to 0 or uniform, weights are sufficient to break symmetry

Activation Functions and Small Gradients

Forward Pass

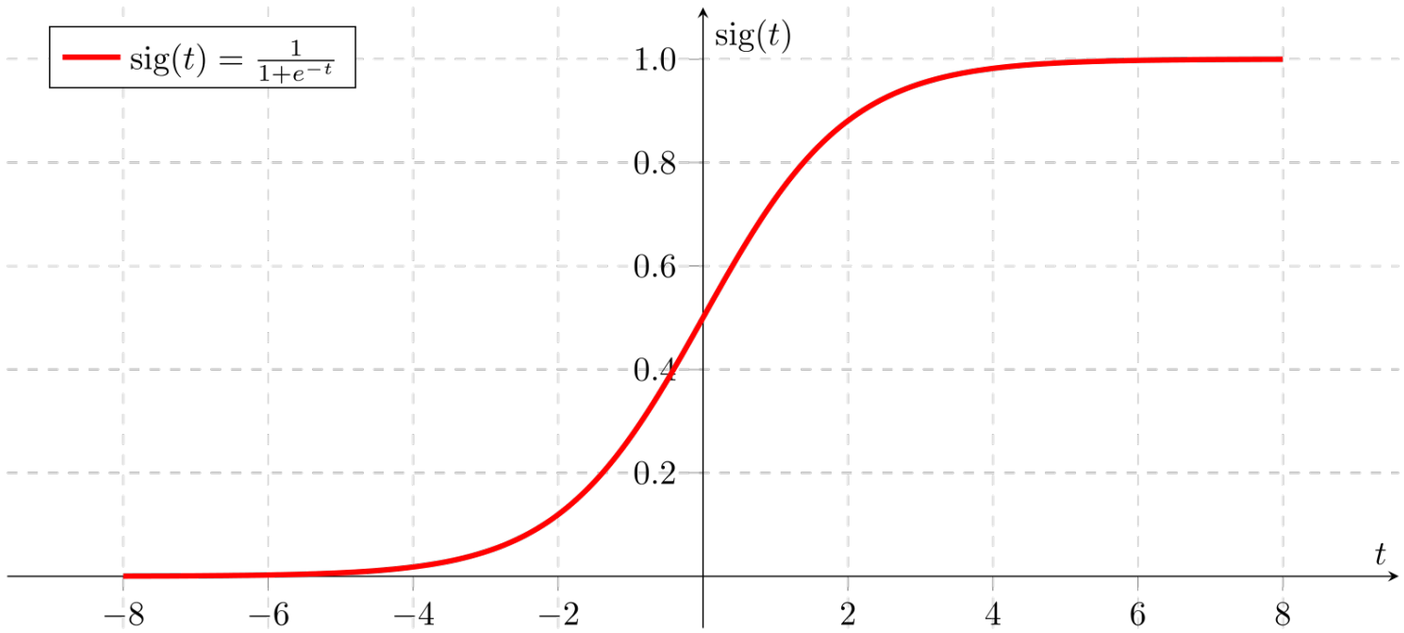
$$x_0 \leftarrow x$$
$$\text{for } j = 0 \text{ to } J-1:$$
$$z_{j+1} \leftarrow W_{j+1} x_j$$
$$x_{j+1} \leftarrow \rho(z_{j+1})$$
$$J = l(x_J, y)$$

Backward Pass

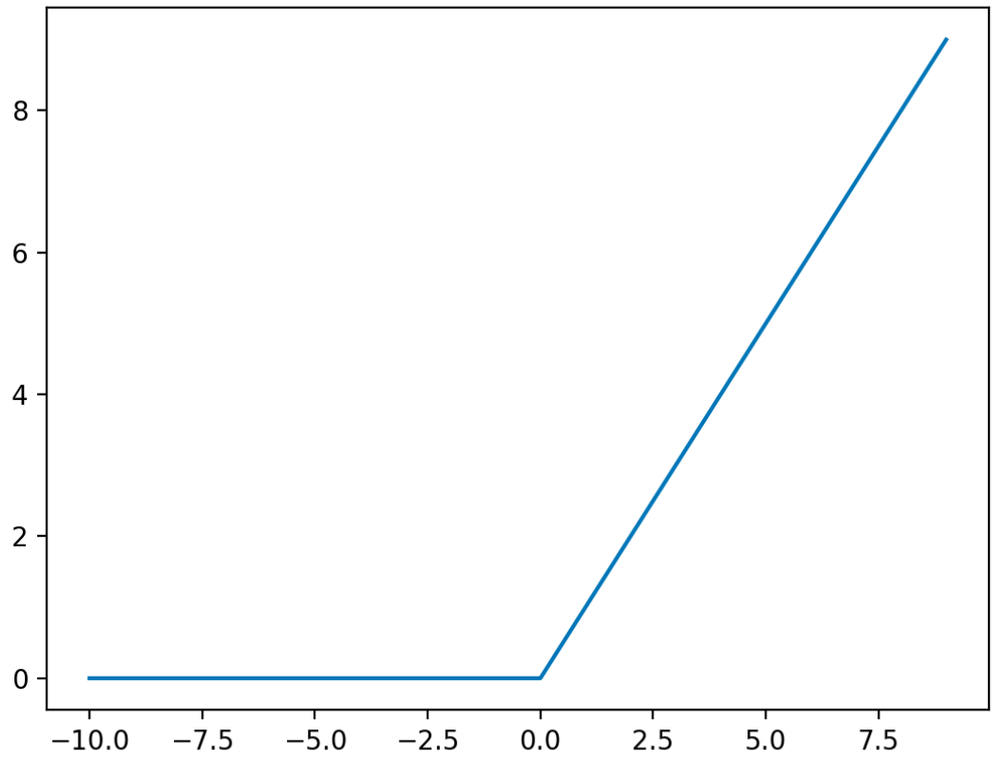
$$v = \nabla_{x_J} L = \nabla_{x_J} l(x_J, y)$$
$$\text{for } j = J-1 \text{ to } 1:$$
$$v \leftarrow \nabla_{z_j} L = v \circ \rho'(z_j)$$
$$\nabla_{W_j} L = v x_{j+1}^T$$
$$v \leftarrow \nabla_{x_j} L = W_j^T v$$

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

$$\rho'(x) = \rho(x) * (1 - \rho(x))$$



Activation Functions and Vanishing Gradients

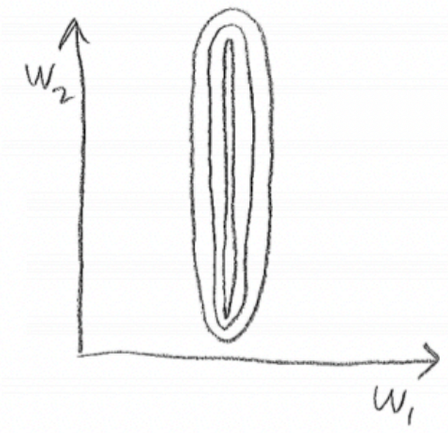


- ReLU non-linearities avoid these saturating regions
- ReLU has become a more popular non-linearity particularly for deeper networks

Input Normalization

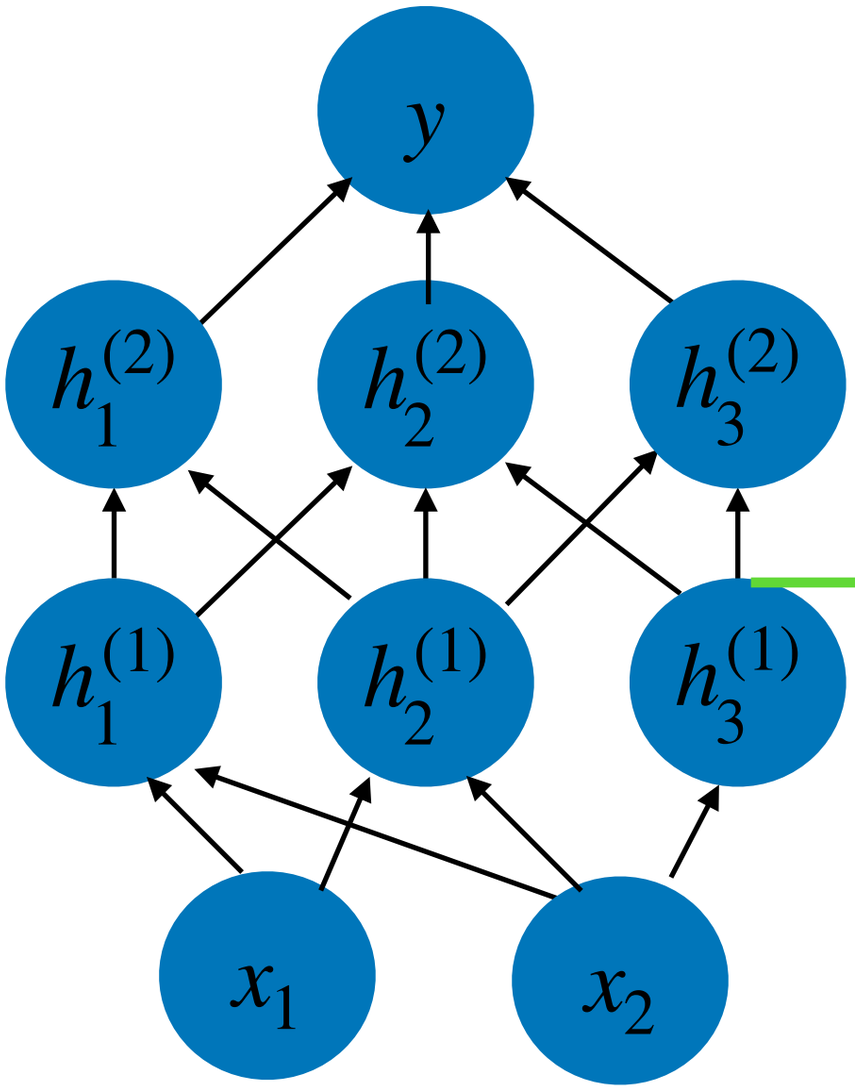
Image Credit: Roger Grosse

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮



- Standardizing the input, 0 mean and unit variance, is a standard pre-processing technique in machine learning
  - 0-mean
  - Unit variance
- Helps with pathological curvature (ill-conditioning) and helpful to understand more complex normalization

Batch Normalization

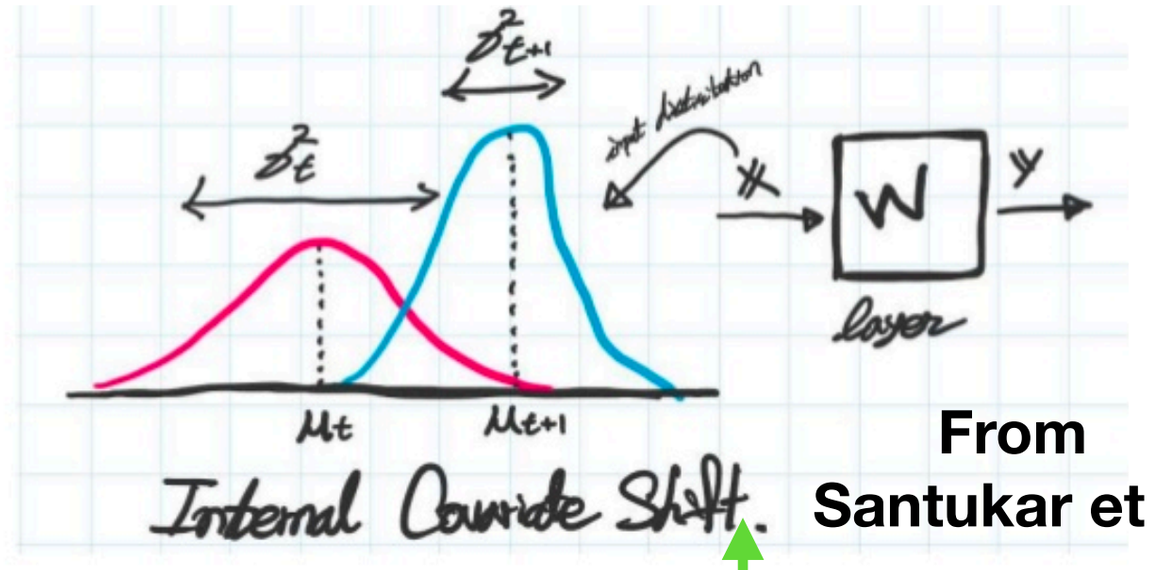


Typically pre-activation

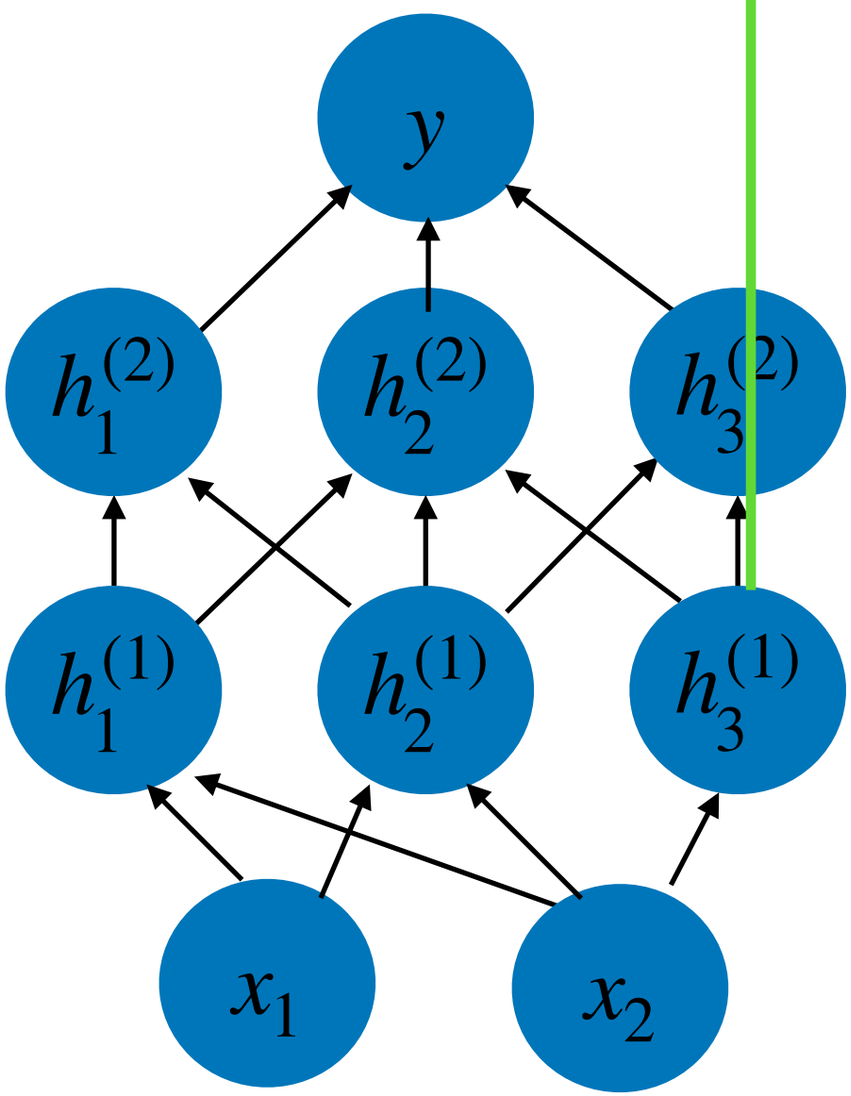
$$a_i^{(j)} = \frac{\gamma}{\hat{\sigma}}(x - \hat{\mu}) + \beta$$

$\hat{\mu}, \hat{\sigma}$ **Estimated from mini-batch**

Batch Normalization



From Santukar et al



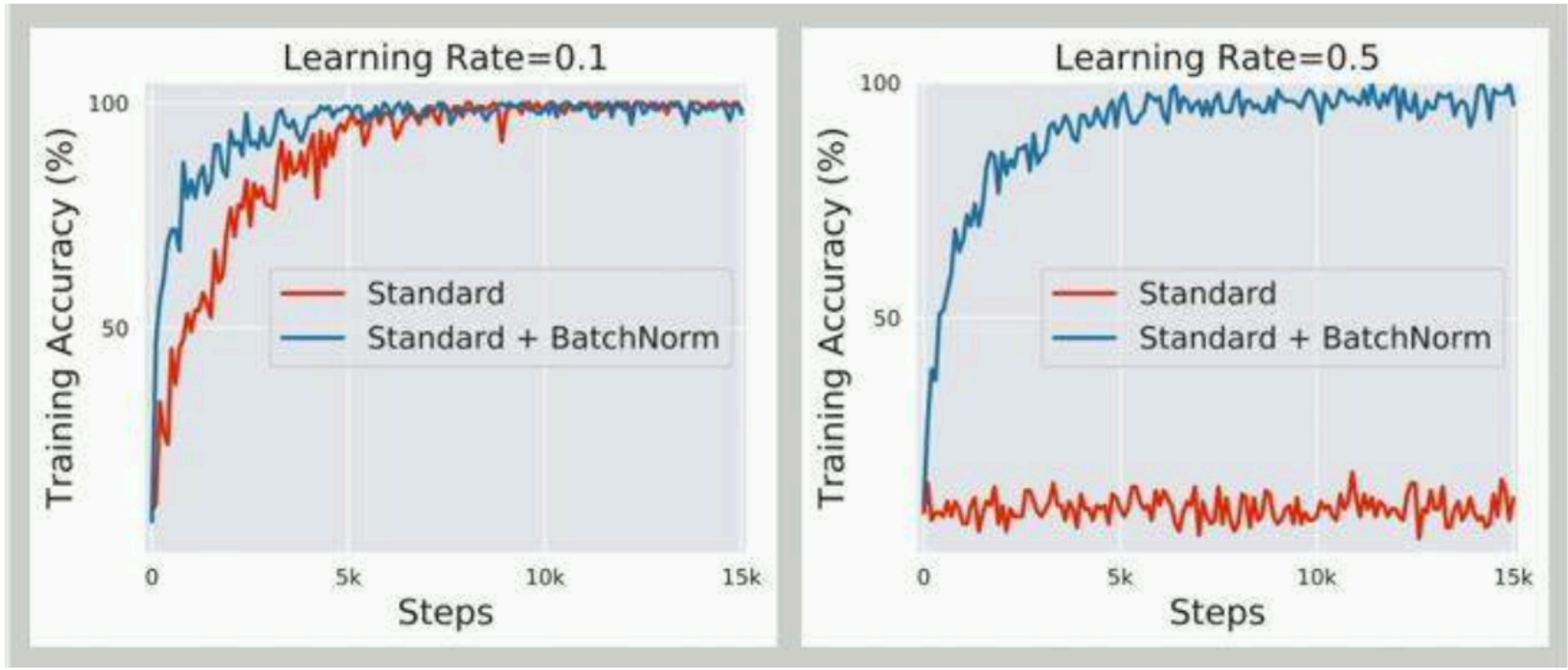
- Originally introduced to battle “internal co-variate shift”
- Has made optimization more robust for variety of models
- Created interest in other normalization techniques
- Most large scale modern networks use some form of normalization now

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

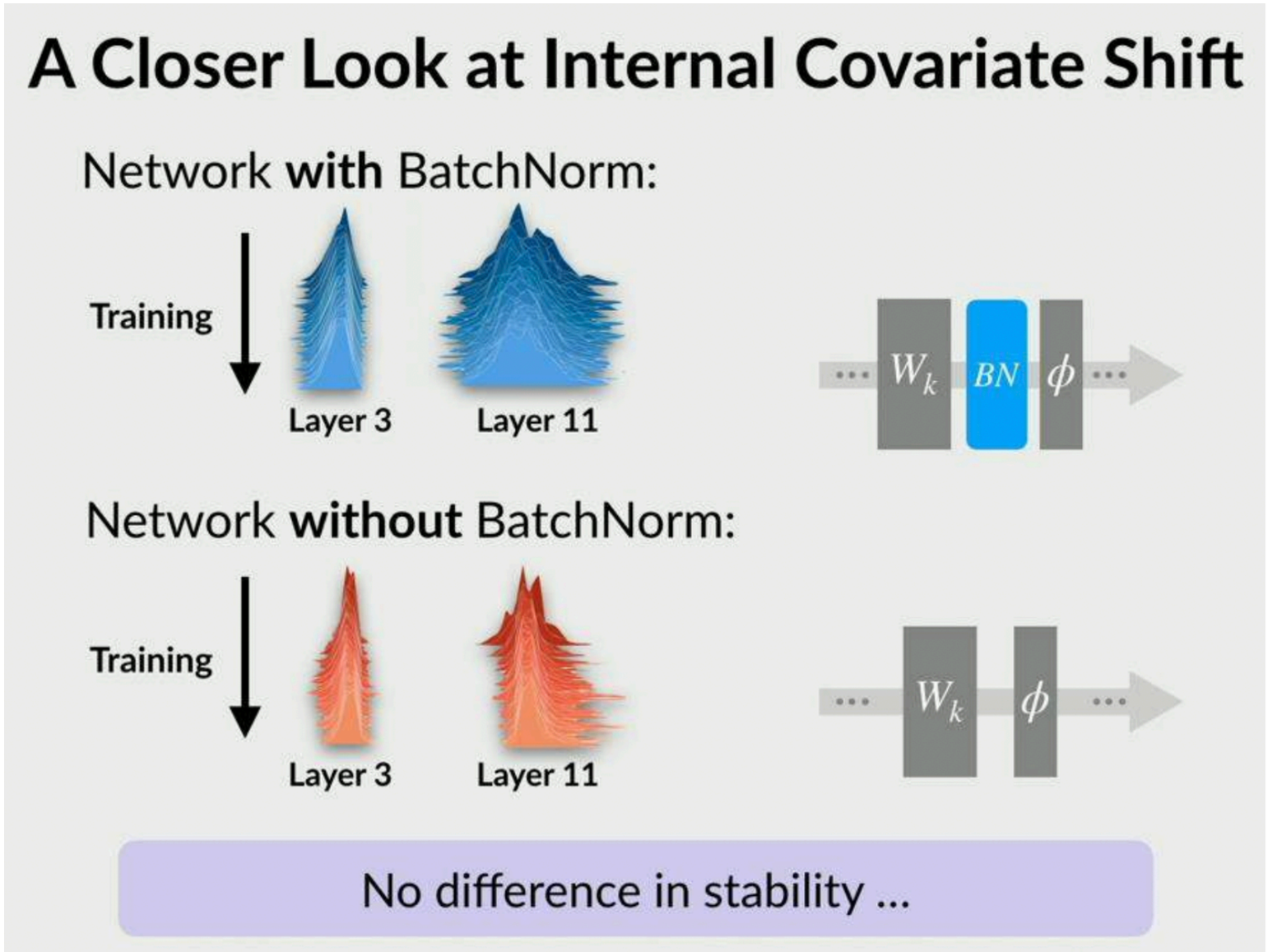
Christian Szegedy
Google Inc., *szegedy@google.com*

Batch Normalization



- Faster convergence
- Robust to hyperparemeters
- Can reach better testing accuracy

Internal Covariate Shift



- Co-variate shift hypothesis has been largely debunked
- Exact nature of batch normalization not so well understood

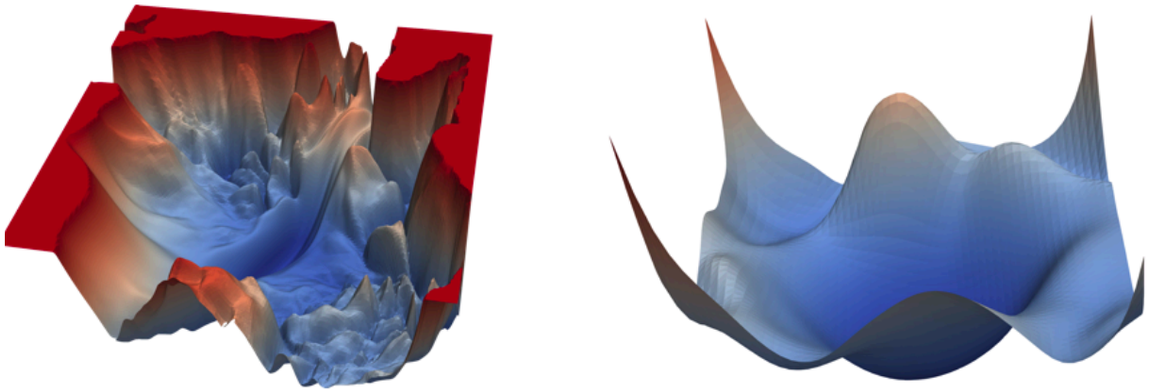
How Does Batch Normalization Help Optimization?

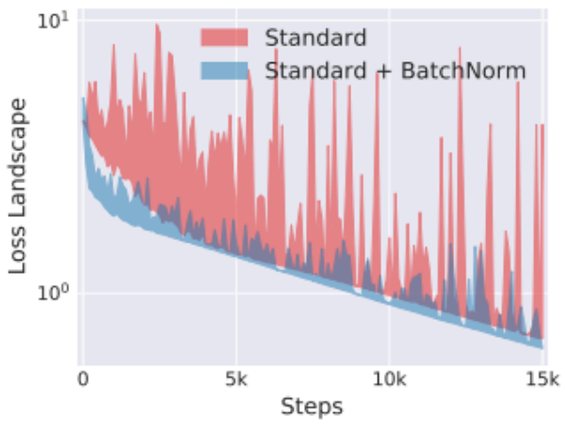
Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Batch Normalization

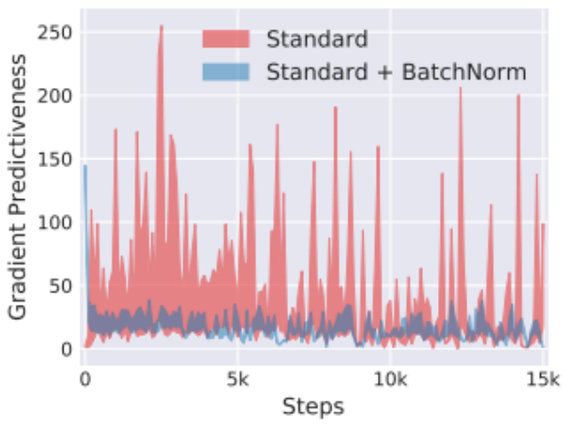
- Santukar et al: BN re-parametrizes the underlying optimization problem to make its landscape significantly more smooth

- Not necessarily the final story on batch norm!





(a) loss landscape



(b) gradient predictiveness

Theorem (*Effect of BatchNorm on the Lipschitzness of the loss*)

For any weights W and loss function L, *we have:*

$$||\nabla_{y_j} L_{BN}||^2 \leq \frac{\gamma^2}{\sigma_j^2}\left(||\nabla_{y_j} L_{Std}||^2 - \mu(\nabla_{y_j} L_{Std})^2 - \frac{1}{m}(\hat{y}_j^\top \nabla_{y_j} L_{Std})^2\right)$$

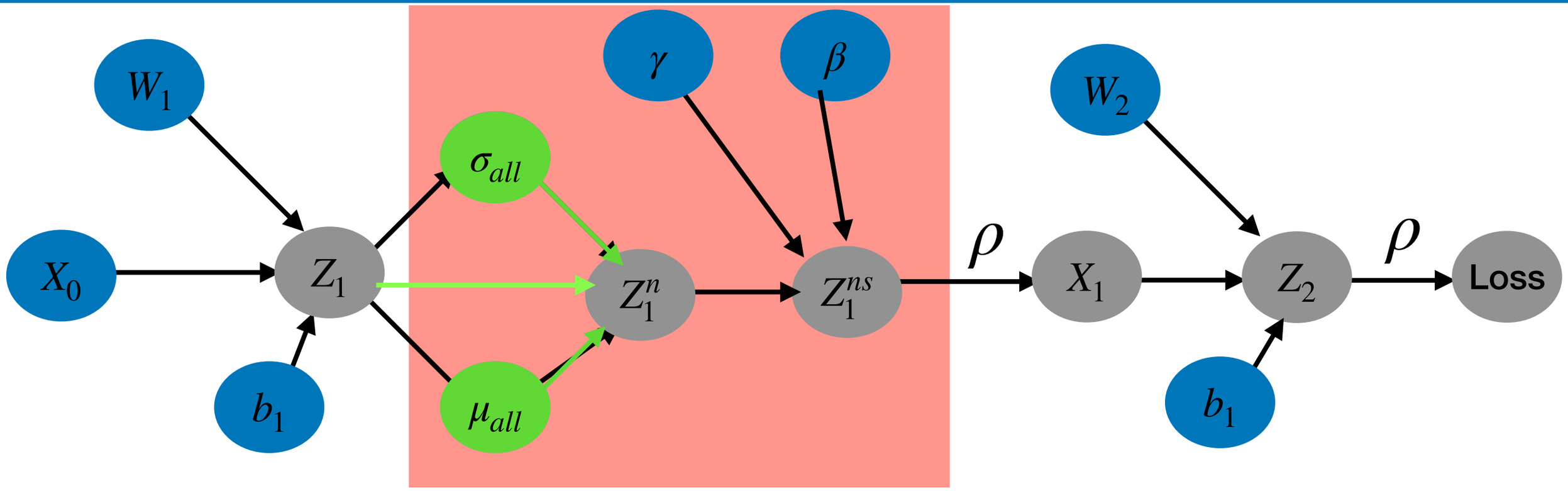
How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Batch Normalization

- Besides accelerating optimization has been observed to act as a regularizer
- Intuition: noise introduced from running mean and variance
- Normalization can reduce sensitivity to initialization
- Critical to Large scale CNNs (2015-recently), GANs, and other hard to train models
- Batch Norm also has other interesting applications in conditional models

Implementing Batch Norm



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

$$\mu_{all} = m * \mu_{all} + (1 - m) * \mu_B$$

$$\sigma_{all} = m * \sigma_{all} + (1 - m) * \sigma_B$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Different Train / Test Behaviour

```
model.train()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(train_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

- Batchnorm must be carefully applied when it comes to eval vs train

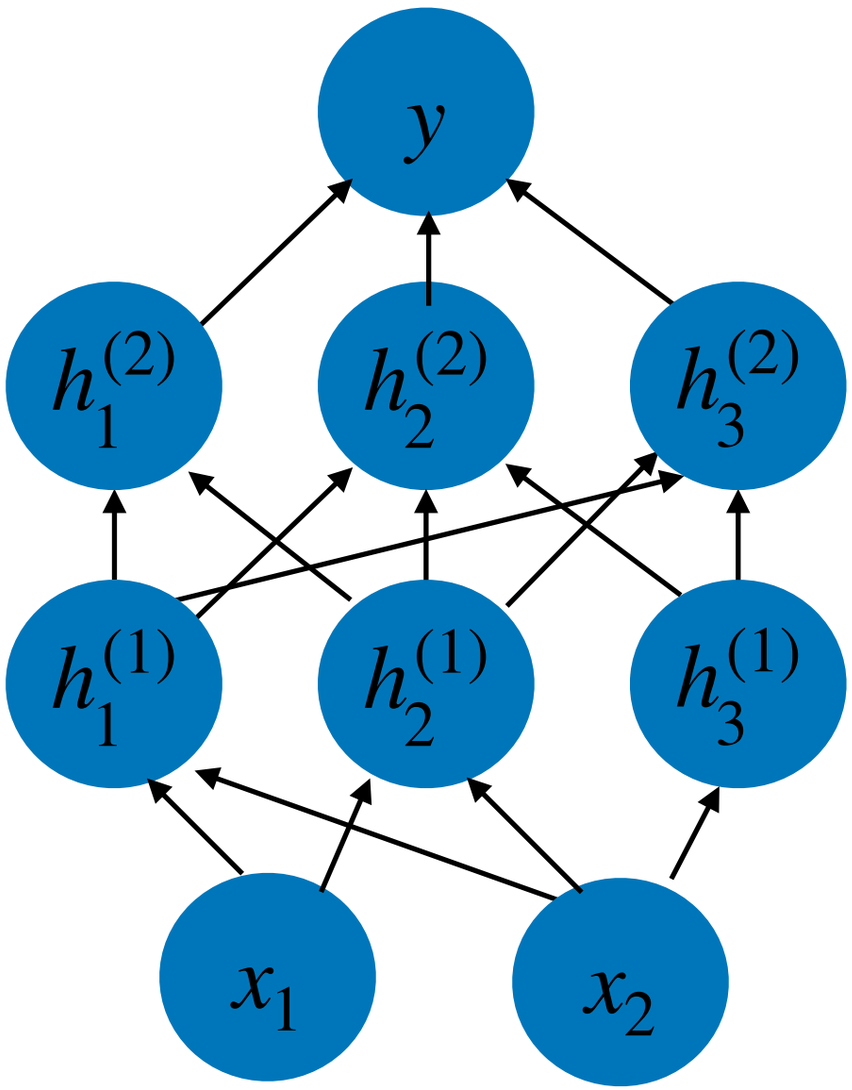
```
model.eval()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(test_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

- Results will not only be bad but invalid/false!!!

Weight Normalization

$$h_i^j = \rho(\mathbf{w}^T \mathbf{h}^{j-1})$$

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$



- Reparametrization of weight
  - Decouples length of weight vector and direction
- Differentiate through normalization

Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks

Tim Salimans
OpenAI
tim@openai.com

Diederik P. Kingma
OpenAI
dpkingma@openai.com

Weight Normalization

$$\nabla_g L = \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{||\mathbf{v}||},$$

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

$$\nabla_{\mathbf{v}} L = \frac{g}{||\mathbf{v}||} M_{\mathbf{w}} \nabla_{\mathbf{w}} L, \quad \text{with} \quad \mathrm{M_w} = \mathrm{I} - \frac{\mathbf{w}\mathbf{w}'}{||\mathbf{w}||^2},$$

- Scales the gradient and projects away from current weight
- Robust to learning rates, v norm adjusts to high learning rates
- Doesn’t introduce dependencies on samples
- No noise as in batch norm — loses regularization?
- Popular in transformer models

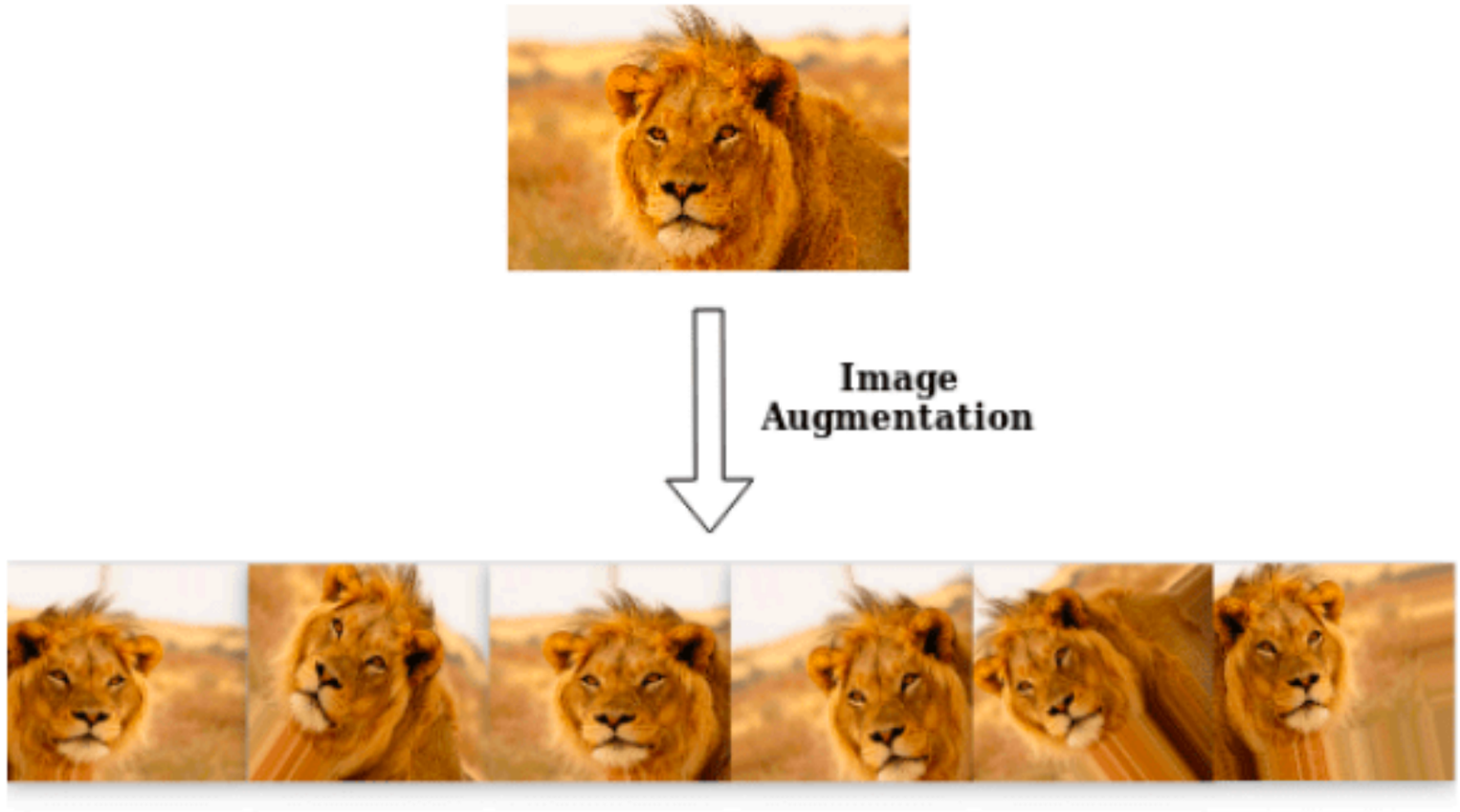
Data Augmentation

- Loosely defined are programmatic ways to modify data to create a larger synthetic dataset

- Allows to introduce domain priors

- Introduced critical noise to avoid overfitting

Data Augmentation



- Essential to high performance of many modern deep learning system
- Allows to capture invariances of the data / introduce domain knowledge
- Reduces overfitting by adding noise to existing samples

Online Data Augmentation

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

trainset_class = CIFAR10(root='.', train=True, download=True, transform=transform_train)
train_loader = torch.utils.data.DataLoader(trainset_class, batch_size=args.batch_size, shuffle=True, num_workers=4)
testset = CIFAR10(root='.', train=False, download=True, transform=transform_test)
val_loader = torch.utils.data.DataLoader(testset, batch_size=1000, shuffle=False, num_workers=2)
```

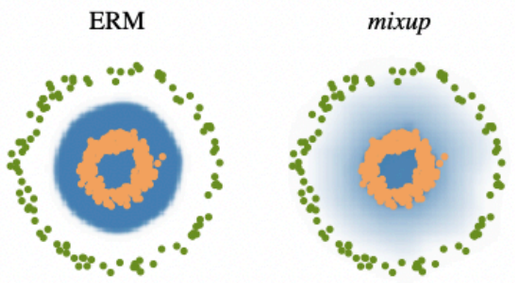
```
model.train()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(train_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()

model.eval()
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(test_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

- Data Augmentation is often applied online
- In large scale dataset with sufficient augmentation randomness we often won’t see exact version 2 times

Data-Dependent Augmentation - Mixup

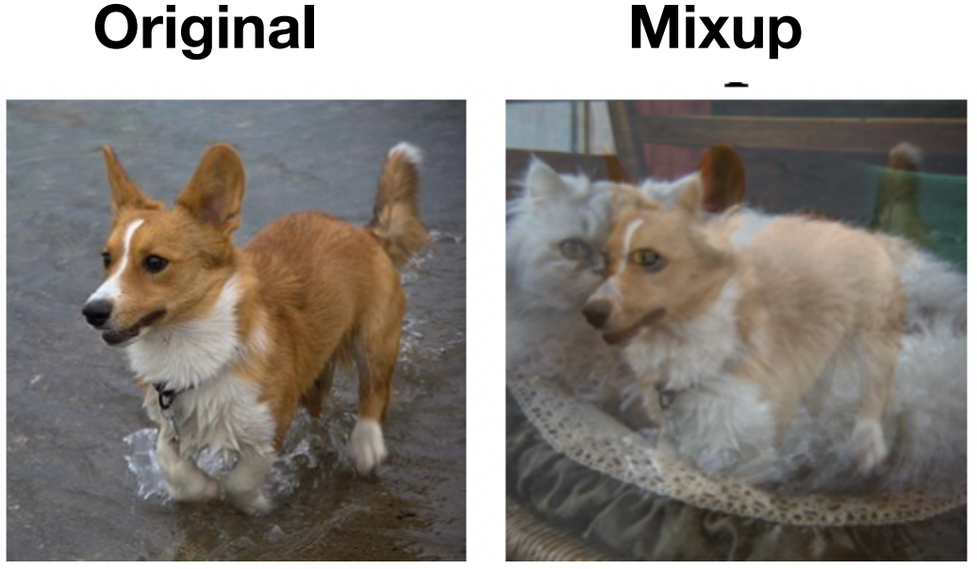
```
# y1, y2 should be one-hot vectors
for (x1, y1), (x2, y2) in zip(loader1, loader2):
    lam = numpy.random.beta(alpha, alpha)
    x = Variable(lam * x1 + (1. - lam) * x2)
    y = Variable(lam * y1 + (1. - lam) * y2)
    optimizer.zero_grad()
    loss(net(x), y).backward()
    optimizer.step()
```

(a) One epoch of *mixup* training in PyTorch.



(b) Effect of *mixup* ($\alpha = 1$) on a toy problem. Green: Class 0. Orange: Class 1. Blue shading indicates $p(y = 1|x)$.

- Mixup
- Manifold Mixup - mixup in intermediate layers



mixup: BEYOND EMPIRICAL RISK MINIMIZATION

Hongyi Zhang
MIT

Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz*
FAIR

Remarks

- Tricks of the trade for learning networks have emerged via lengthy mix of engineering and science

- Many of them, even relatively simple ones, are still not 100% understood or have been shown to act differently than the originally motivated
  - Batchnorm
  - Weight Decay
  - Data Augmentation

- Interaction between all these elements and their effect on learning dynamics is non-trivial

SGD+Backprop

- In the 1980s it was unclear what is a good method to learn neural network parameters

- Many fundamental components in modern deep learning pipelines are potentially relics related to gradient based learning

  - Vanishing gradient motivated many components of models - violation of optimization/objective barrier

- Other approaches may lead to breakthroughs in efficiency of learning, parallelization, and open up new architecture possibilities

- Difficult to get traction, the “momentum” is high for SGD+backprop due to existing tool boxes that works at scale

  - architecture elements, hyper-parameters, and software frameworks entrenched

- Biological plausibility

Evolutionary Algorithms

Optional

- Genetic algorithms evolve a population of N individual Networks at each step we evaluate the fitness of the parameters

- The top performing are selected for “mutation”

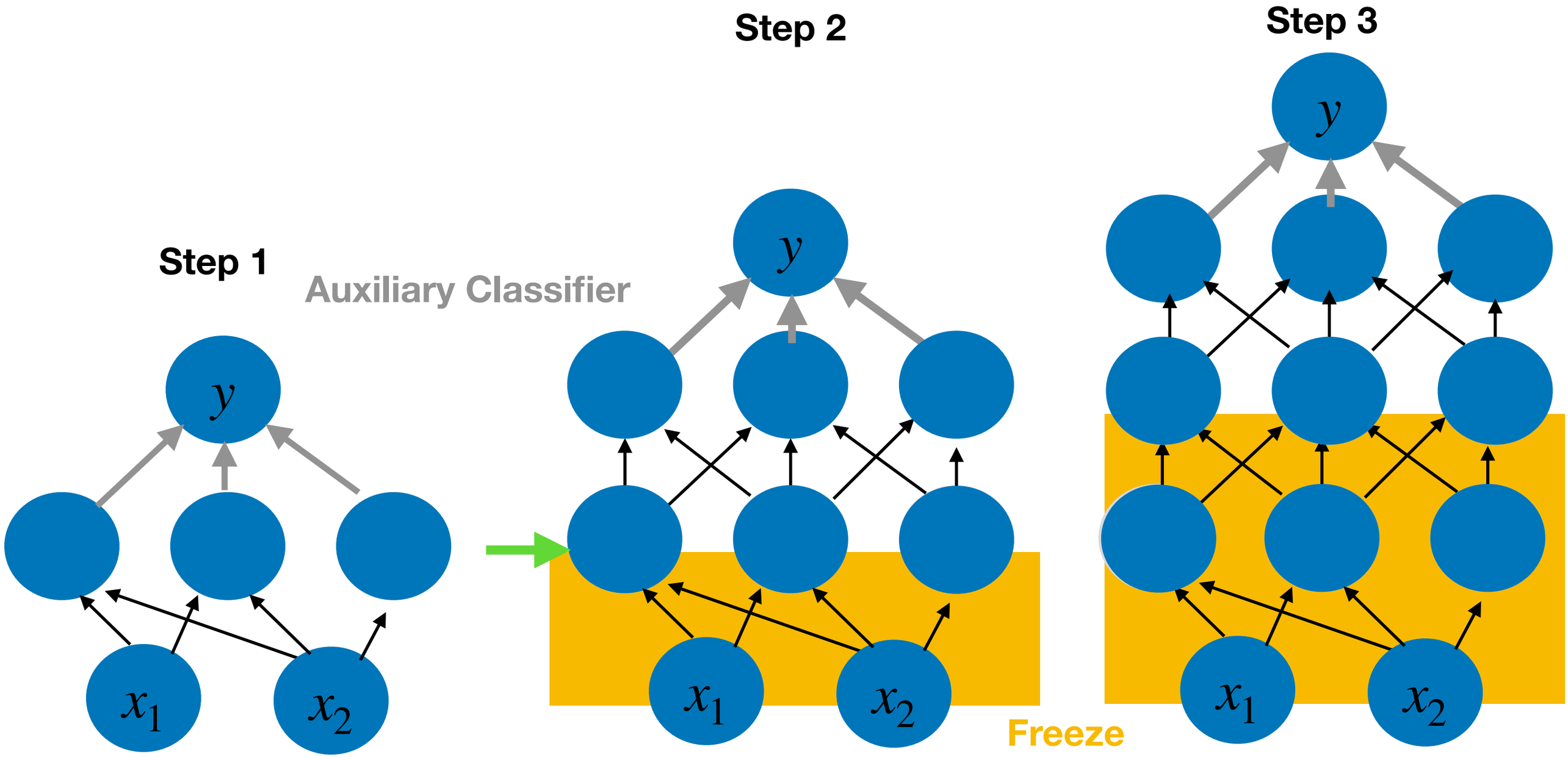
e.g. $w_2 = w_1 + \sigma\epsilon \qquad \epsilon \sim N(0,I)$

- Sometimes combined in hybrid gradient based learning systems
  - Useful in non-differentiable cases
  - Evolve architecture
- Convergence guarantees and theory for EV algorithms is sparse
- For large models they are currently very inefficient

Local Learning

Optional

- Classic algorithm (1965) that sequentially learns each layer
- Recently shown to yield strong performance on large scale data results with interesting properties



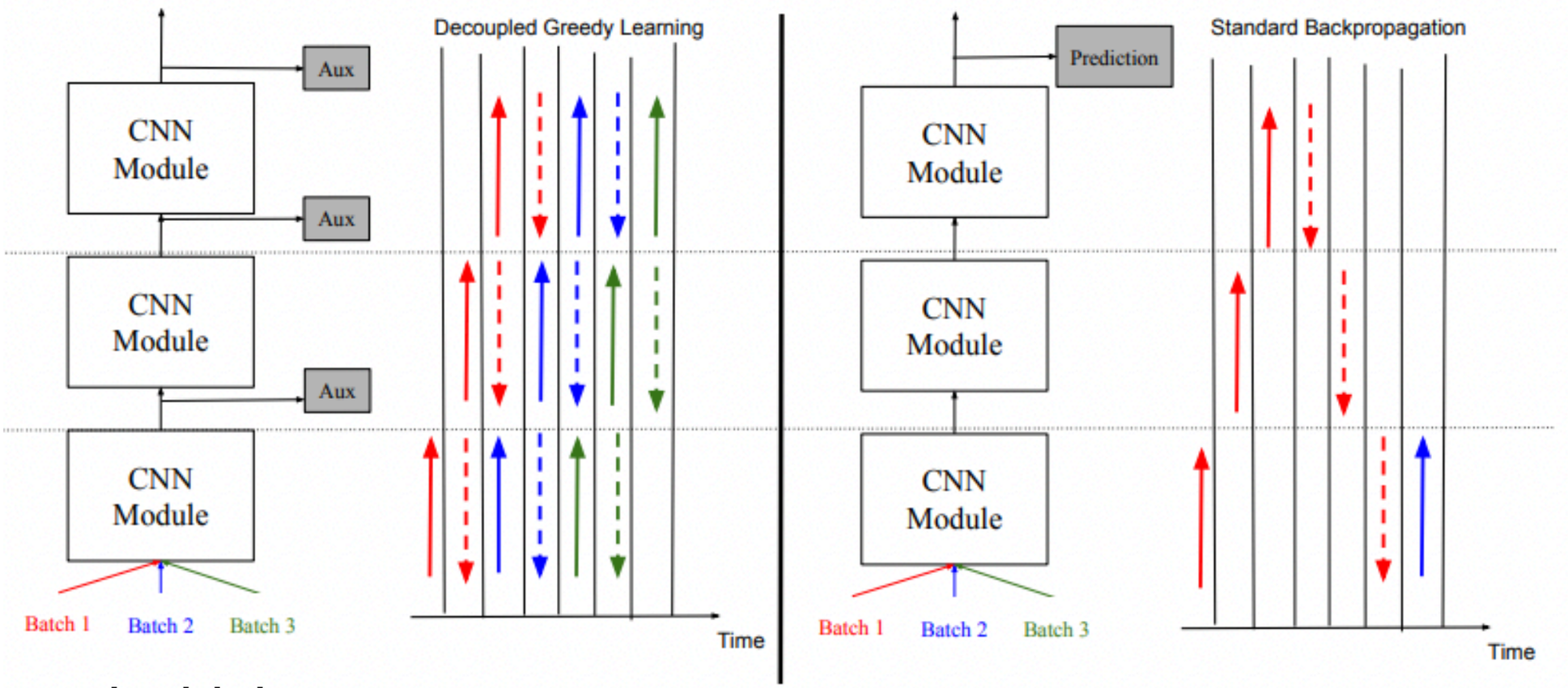
Ivakhnenko, A. G. and Lapa, V. Cybernetic predicting devices. CCM Information Corporation, 1965.

Belilovsky, Eugene, Michael Eickenberg, and Edouard Oyallon. "Greedy layerwise learning can scale to imagenet." *ICML* 2019

Local Parallel Learning

Optional

- Parallel version of this algorithm has recently been shown
- Recently shown to yield strong results without end-to-end learning



Simultaneously minimize

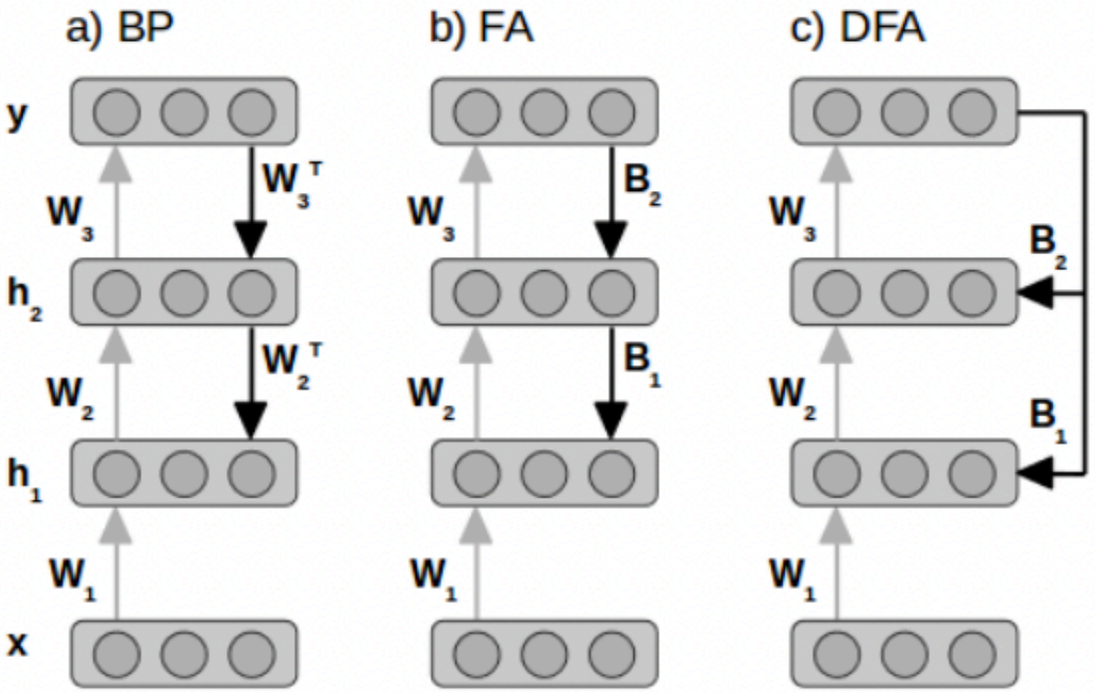
$$\min_{\theta_j,\gamma_j} \hat{\mathcal{L}}(\boldsymbol{X}_j, Y; \theta_j, \gamma_j),$$

Nøkland, Arild and Lars Hiller Eidnes. “Training Neural Networks with Local Error Signals.” *ICML* (2019).

Belilovsky, Eugene et al. “Decoupled Greedy Learning of CNNs.” *ArXiv* abs/1901.08164 (2019): n. pag.

FeedBack Alignment

Optional



Backward Pass

$$v = \nabla_{x_J} L = \nabla_{x_J} l(x_J, y)$$

for $j = J - 1$ to 1 :

$$v \leftarrow \nabla_{z_j} L = v \circ \rho'(z_j)$$

$$\nabla_{W_j} L = v x_{j+1}^T$$

$$\boxed{v \leftarrow \nabla_{x_j} L = W_j^T v}$$

Feedback Alignment Use a random matrix B_j

$$v \leftarrow \nabla_{x_j} L = B_j v$$

Direct Feedback Alignment Use $v = \nabla_{x_J} l(x_J, y)$

Random synaptic feedback weights support error backpropagation for deep learning

Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed & Colin J. Akerman

Nature Communications **7**, Article number: 13276 (2016) | Cite this article

Direct Feedback Alignment Provides Learning in Deep Neural Networks

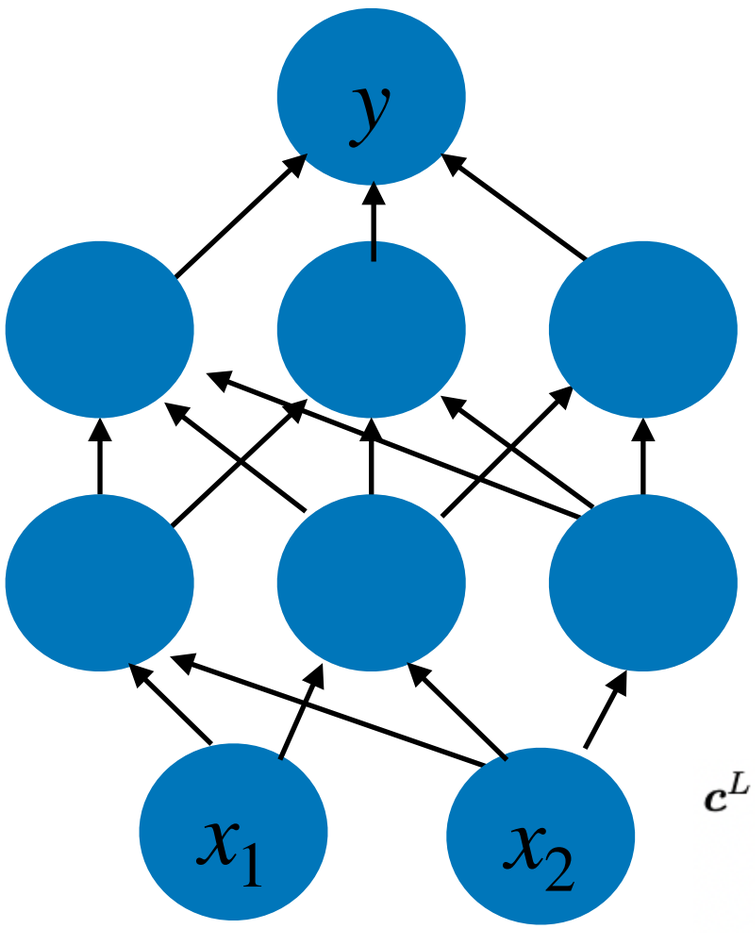
Arild Nøkland
Trondheim, Norway
arild.nokland@gmail.com

ADMM

Optional

$$\min_{\boldsymbol{W}} \quad \sum_{t=1}^{n} \mathcal{L}(\boldsymbol{y}_t, \boldsymbol{a}_t^L, \boldsymbol{W}^{L+1}),\ where\ \boldsymbol{a}_t^l = \sigma_l(\boldsymbol{c}_t^l),$$
$$s.t.\ \boldsymbol{c}_t^l = \boldsymbol{W}^l \boldsymbol{a}_t^{l-1},\ l = 1, ..., L,\ and\ \ \boldsymbol{a_t}^0 = \boldsymbol{x}_t. (1)$$

$$f(\boldsymbol{W}, \boldsymbol{C}) = \sum_{t=1}^{n} \mathcal{L}(y_t, \sigma_L(\boldsymbol{c}_t^L), \boldsymbol{W}^{L+1}) + \mu \sum_{t=1}^{n} \sum_{l=1}^{L} ||\boldsymbol{c}_t^l - \boldsymbol{W}^l \sigma_{l-1}(\boldsymbol{c}_t^{l-1})||_2^2.$$



while more samples **do**
  Input $(\boldsymbol{x}_t, y_t)$
  $\boldsymbol{C} \leftarrow$ **encodeInput**($\boldsymbol{x}_t$, $\boldsymbol{W}_{t-1}$) % forward: compute linear activations at layers $1, ..., L$
  $\boldsymbol{C} \leftarrow$ **updateCodes**($\boldsymbol{C}$, y_t, $\boldsymbol{W}_{t-1}$, μ) % backward: error propagation by activation (code) changes
  $\boldsymbol{W}_t \leftarrow$ **updateWeights**($\boldsymbol{W}_{t-1}$, $\boldsymbol{x}_t$, y_t, $\boldsymbol{C}$, μ, η, Mem)
end while

$$\boldsymbol{c}^L = \arg\min_{\boldsymbol{c}} \mathcal{L}(y, \sigma_L(\boldsymbol{c}), \boldsymbol{W}^{L+1}) + \mu ||\boldsymbol{c} - \boldsymbol{W}^L \sigma_{L-1}(\boldsymbol{c}^{L-1})||_2^2$$

$$\boldsymbol{c}^l = \arg\min_{\boldsymbol{c}} \mu ||\boldsymbol{c}^{l+1} - \boldsymbol{W}^{l+1} \sigma_l(\boldsymbol{c})||_2^2 + \mu ||\boldsymbol{c} - \boldsymbol{W}^l \sigma_{l-1}(\boldsymbol{c}^{l-1})||_2^2,$$

Beyond Backprop: Online Alternating Minimization with Auxiliary Variables

Anna Choromanska*[1] Benjamin Cowen*[1] Sadhana Kumaravel*[2] Ronny Luss*[2] Mattia Rigotti*[2]
Irina Rish*[2] Brian Kingsbury[2] Paolo DiAchille[2] Viatcheslav Gurev[2] Ravi Tejwani[3] Djallel Bouneffouf[2]