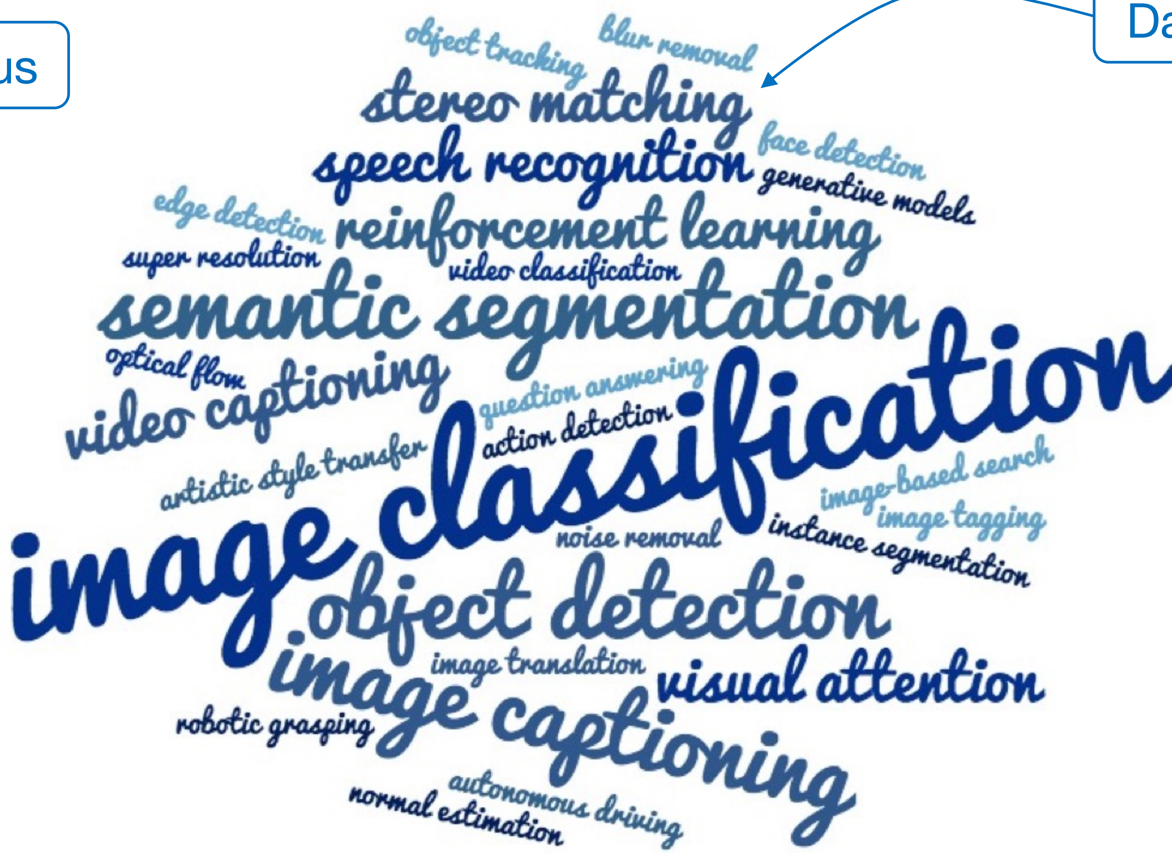
The Effect of Batch Normalization on Deep Convolutional Neural Networks

Fabian Schilling

Convolutional networks are versatile

My focus

Dario's thesis

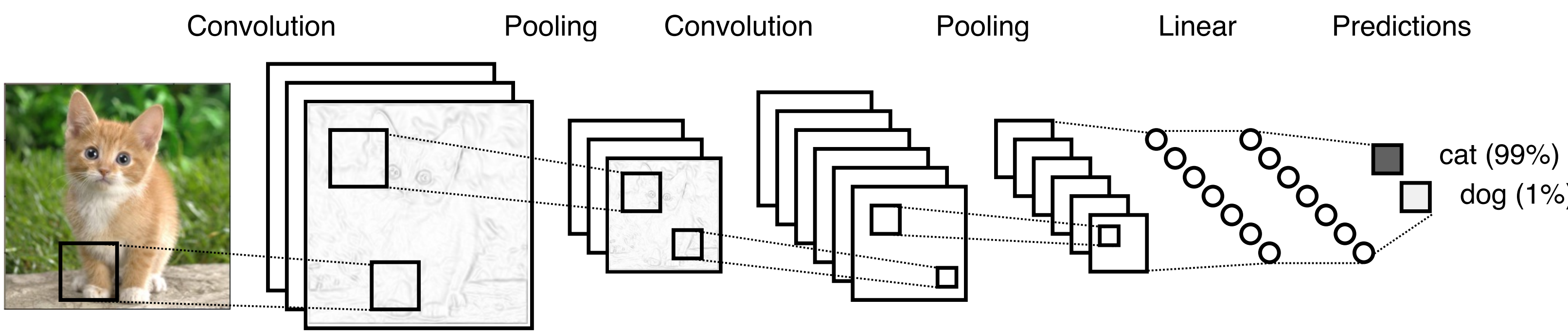


Results apply to all of these!

Background & theory

What are the prerequisites for batchnorm?

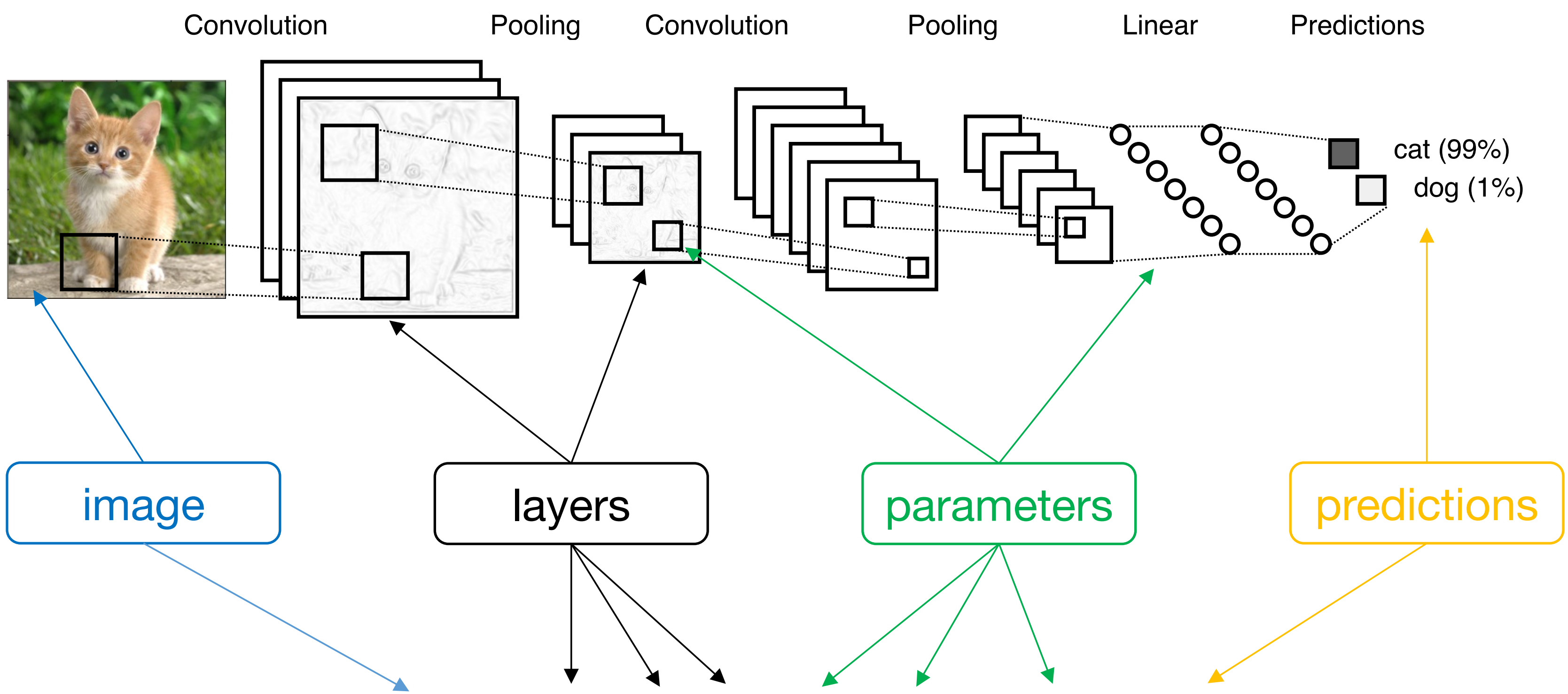
Overview of convnet architecture



$$=$$

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = f_l(\dots f_2(f_1(\boldsymbol{x}, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2) \dots), \boldsymbol{\theta}_l) = \hat{\boldsymbol{y}}$$

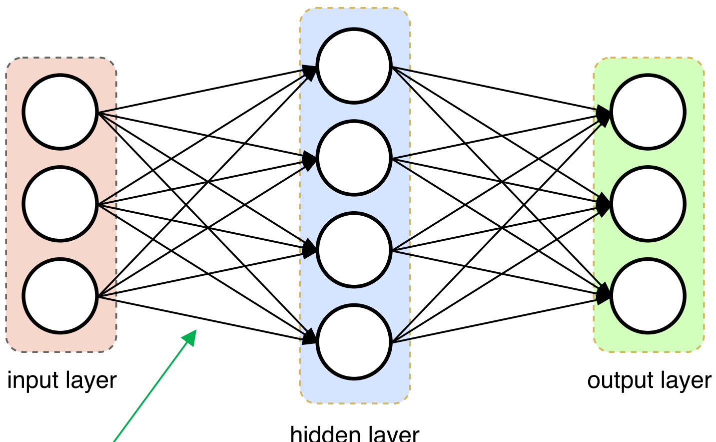
Overview of convnet architecture



$$f(\boldsymbol{x},\boldsymbol{\theta}) = f_l(\dots f_2(f_1(\boldsymbol{x},\boldsymbol{\theta}_1),\boldsymbol{\theta}_2)\dots),\boldsymbol{\theta}_l) = \hat{\boldsymbol{y}}$$

Linear and convolutional layer

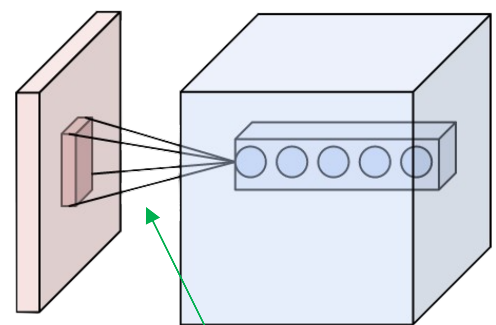
Linear layer



matrix multiplication!

$$\mathrm{linear}(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b}$$

Convolutional layer



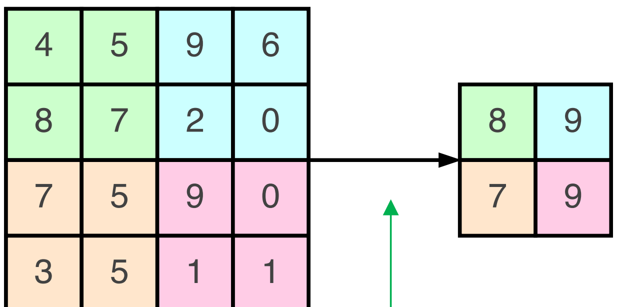
discrete convolution!

$$\mathrm{conv}(\boldsymbol{x}) = \sum_{u,v} \boldsymbol{x}_{uv} * \boldsymbol{w} + \boldsymbol{b}$$

Figures adapted from CS231n: convolutional neural networks for visual recongition (http://cs231n.stanford.edu/)

Pooling and activation layer

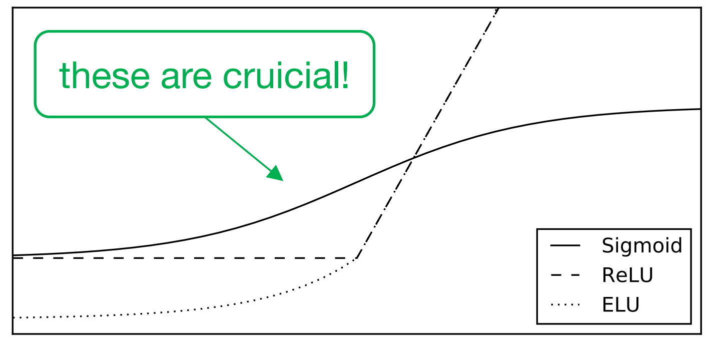
Pooling layer



just downsampling!

$$\mathrm{maxpool}(\boldsymbol{x}) = \max_{u,v} \boldsymbol{x}_{uv}$$

Activation layer

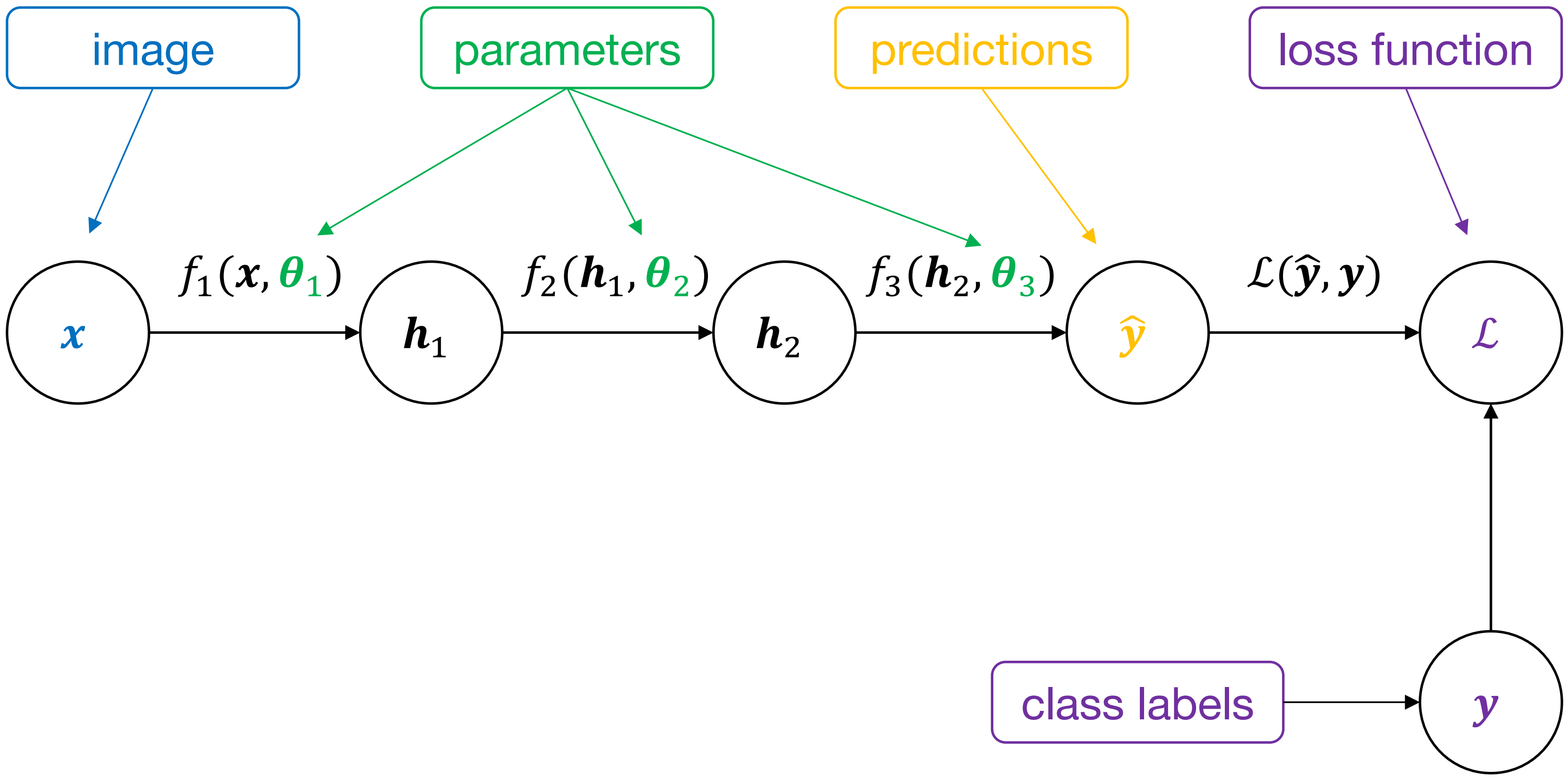


$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

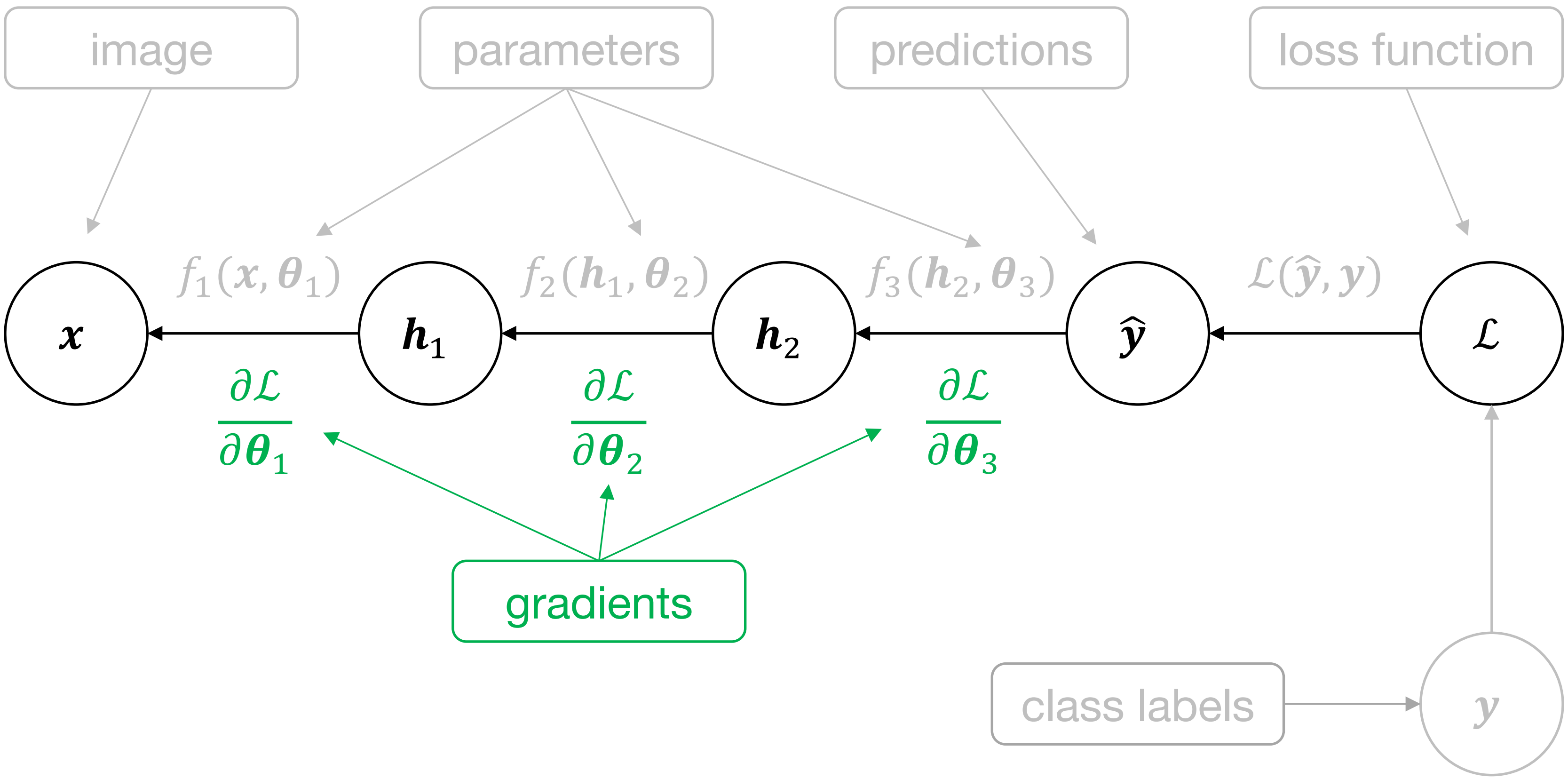
$$\mathrm{ReLU}(x) = \max(x, 0)$$

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

Backpropagation: forward pass



Backpropagation: backward pass



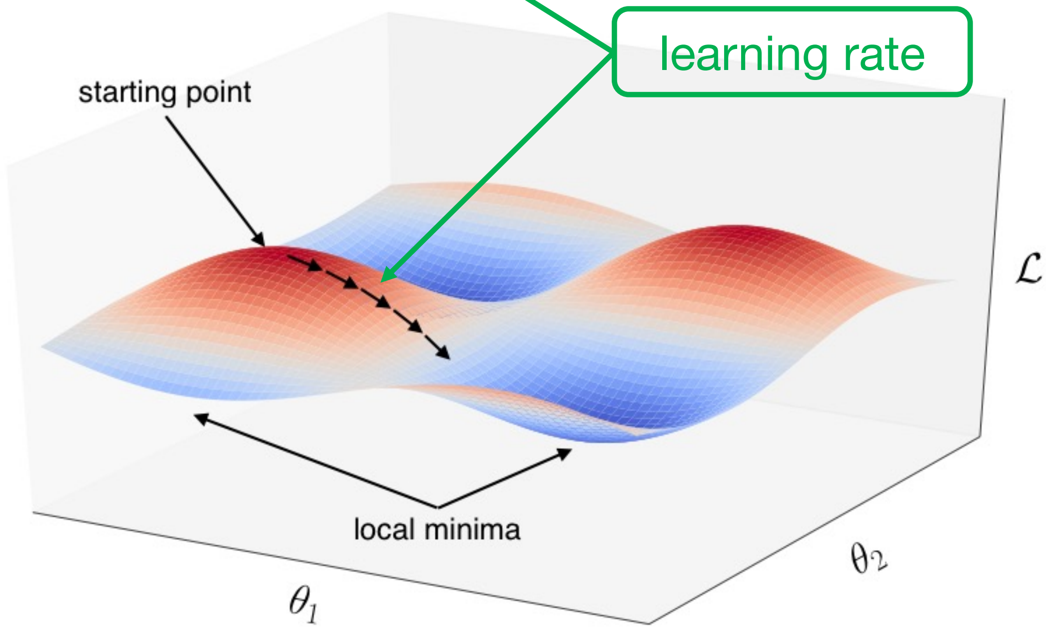
key takeaway: backprop works for any (differentiable) computational DAG!

Mini-batch stochastic gradient descent

batch size

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \frac{1}{\mathcal{B}} \sum_{i=1}^{\mathcal{B}} \frac{\partial \mathcal{L}(\widehat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \boldsymbol{\theta}}$$

learning rate

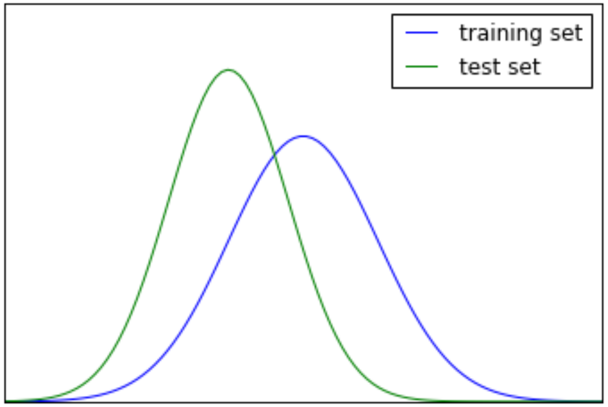


Why use mini-batches?

- **Memory constraints**, entire dataset might not fit
- **Time constraints**, can't parallelize one example
- Noisy updates help escape local minima!

The problem: internal covariate shift

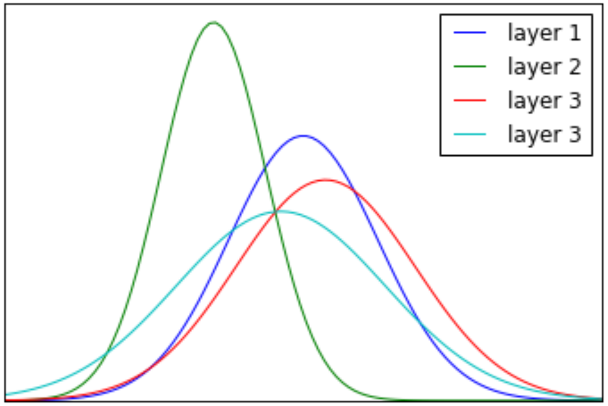
Covariate shift



Distribution of training and test set different

solution: data preprocessing

Internal covariate shift *



Distributions of **individual network layers** change during training

solution: batchnorm!

* Ioffe et. al "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" (2015)

The solution: batch normalization*

- Subtract batch mean $\mu_{\mathcal{B}}$
- Divide by batch standard deviation $\sigma_{\mathcal{B}}$
- Add ϵ for numerical stability

$\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are replaced with population statistics at test time!

$$\text{batchnorm}(x) = \gamma \cdot \frac{x - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta$$

- Scale and shift parameters γ and β can be learned with backprop
- Batch normalization is differentiable!

* Ioffe et. al "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" (2015)

Objective: the effect of batchnorm

- What **general** effects does batchnorm have on…
  - …classification accuracy?
  - …convergence speed?

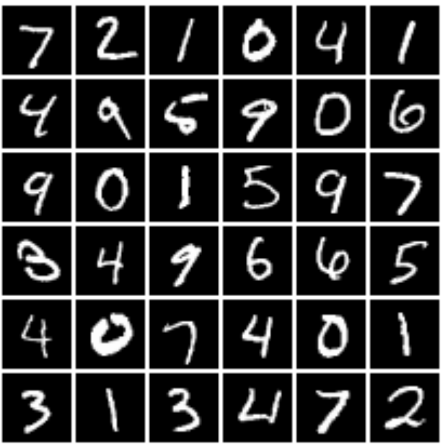
- What **specific** effects does batchnorm have on...
  - …activation functions?
  - …learning rates?
  - …weight initialization?
  - …regularization methods?
  - …batch sizes?
  - …wall time?

Experimental setup

How did we conduct our experiments?

Datasets used in experiments

MNIST

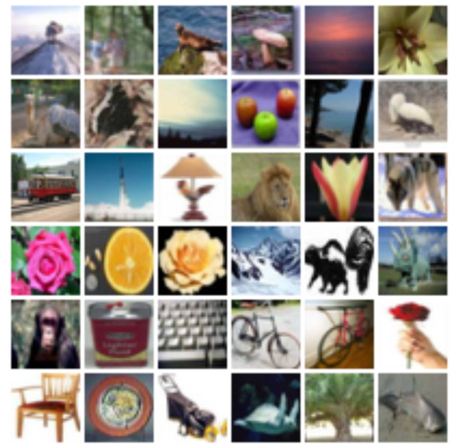


- Handwritten digits
- **32x32 grayscale!**
- 10 classes
- Approx. balanced
- 60k /10k split
- 99.8% best acc.

SVHN

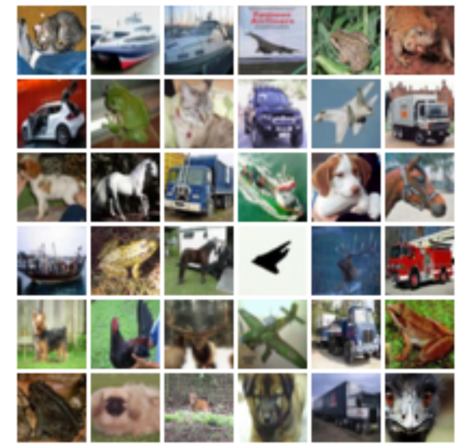
- Housing numbers
- 32x32 color
- 10 classes
- **Unbalanced!**
- 73k / 26k split
- 98.3% best acc.

CIFAR10



- Objects & animals
- 32x32 color
- 10 classes
- Perfectly balanced
- 50k / 10k split
- 96.5% best acc.

CIFAR100



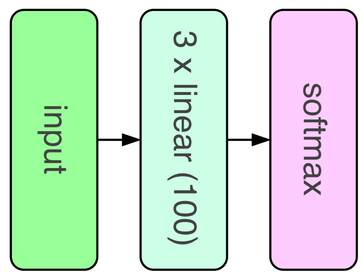
- Objects & animals
- 32x32 color
- **100 classes!**
- Perfectly balanced
- 50k / 10k split
- 75.7% best acc.

increasing difficulty!

Models used in the experiments

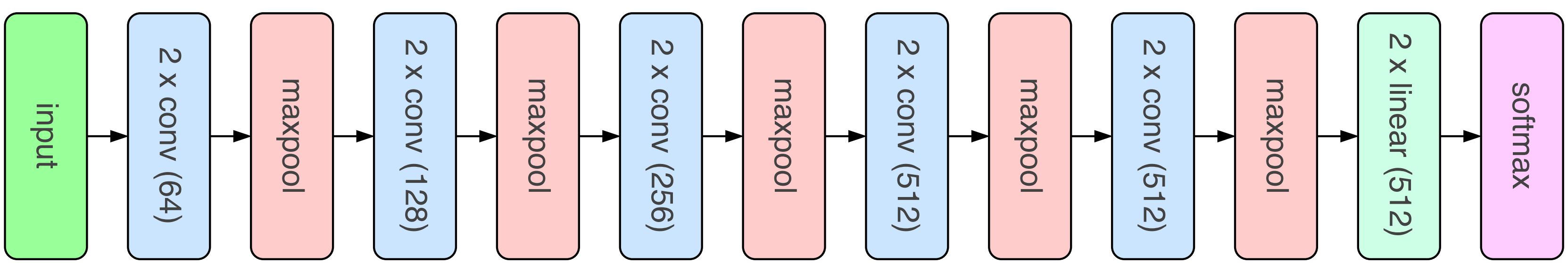
toy model for initial experiments

Multi-layer perceptron:



we will focus on this one!

Deep convolutional neural network:

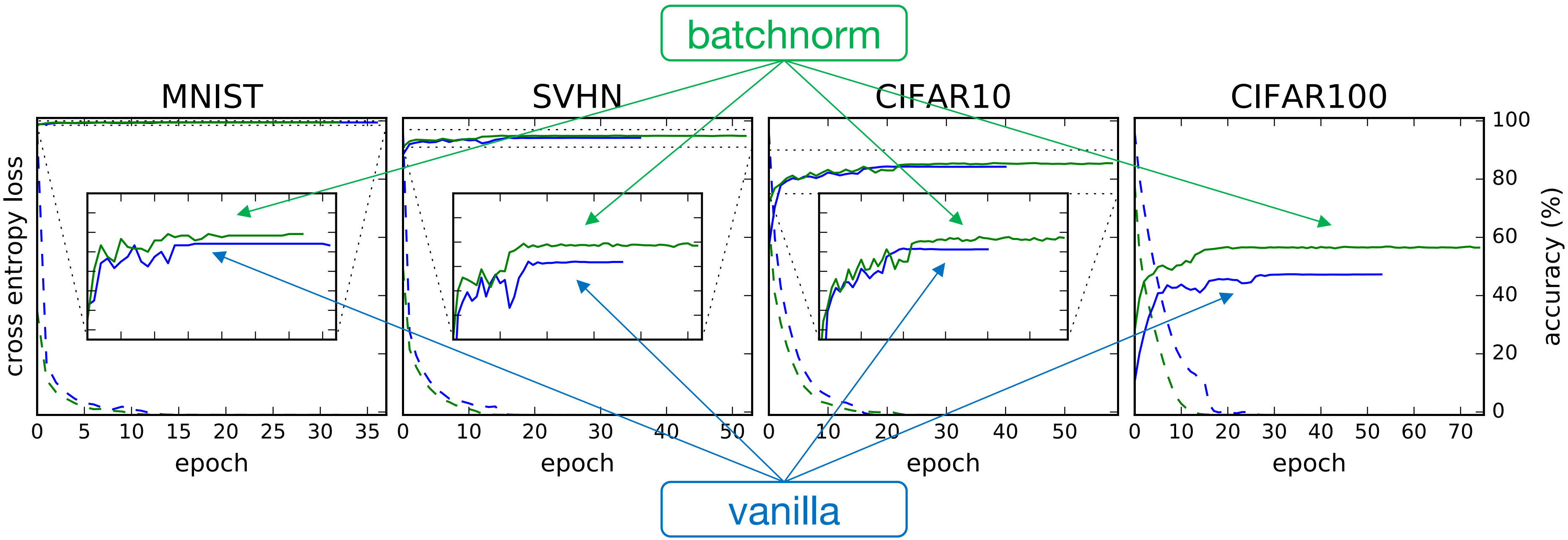


Activation function after each convolution & linear layer, 3x3 kernels and zero padding for convolutions, 2x2 max-pooling

Experimental results

What did we find out?

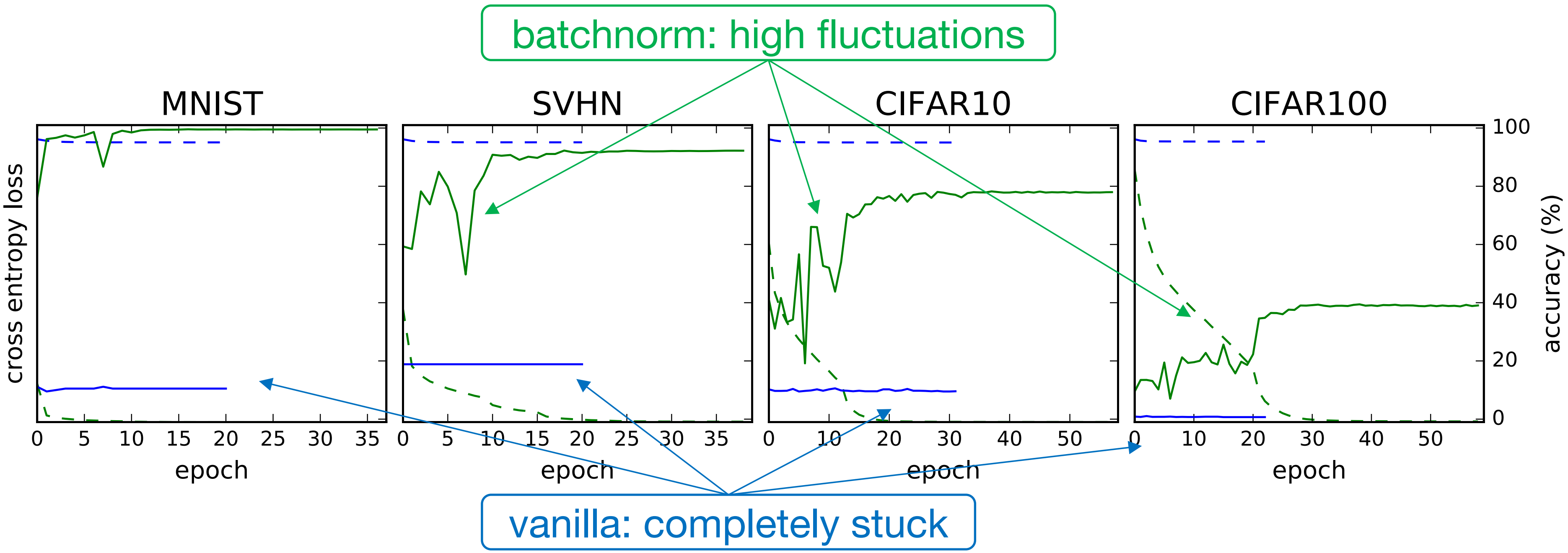
Batchnorm achieves higher accuracies



- **Baseline model** with ReLU activations and Kaiming* initialization
- Batchnorm layer added before all activation functions
- Near-zero losses on all datasets (overfitting)
- Batchnorm beats the vanilla network on all datasets!
- Score: Batchnorm 1 – vanilla 0

* Kaiming He et. al: “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification” (2015)

Batchnorm can handle Sigmoids

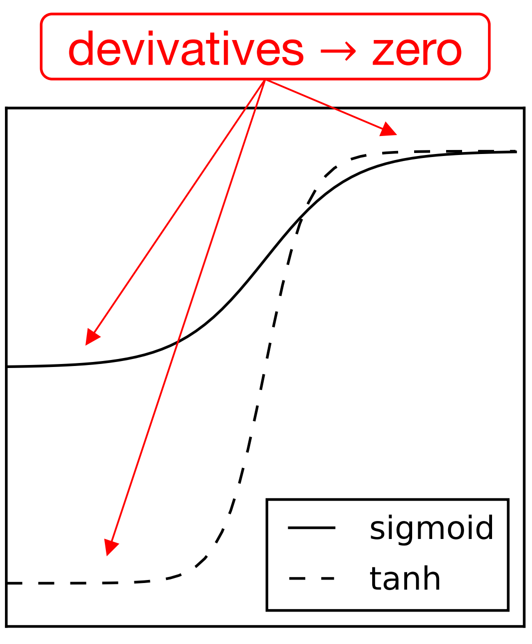


- Exchange ReLU nonlinearities with Sigmoids (and Xavier initialization)
- Vanilla network is stuck at random guessing and cannot escape
- Batchnorm model converges but with major fluctuations
- Score: Batchnorm 2 – vanilla 0
- Why does this happen to the vanilla model?

* Xavier Glorot et. al: “Understanding the Difficulty of Training Deep Feedforward Neural Networks” (2010)

A sidenote on saturating nonlinearities

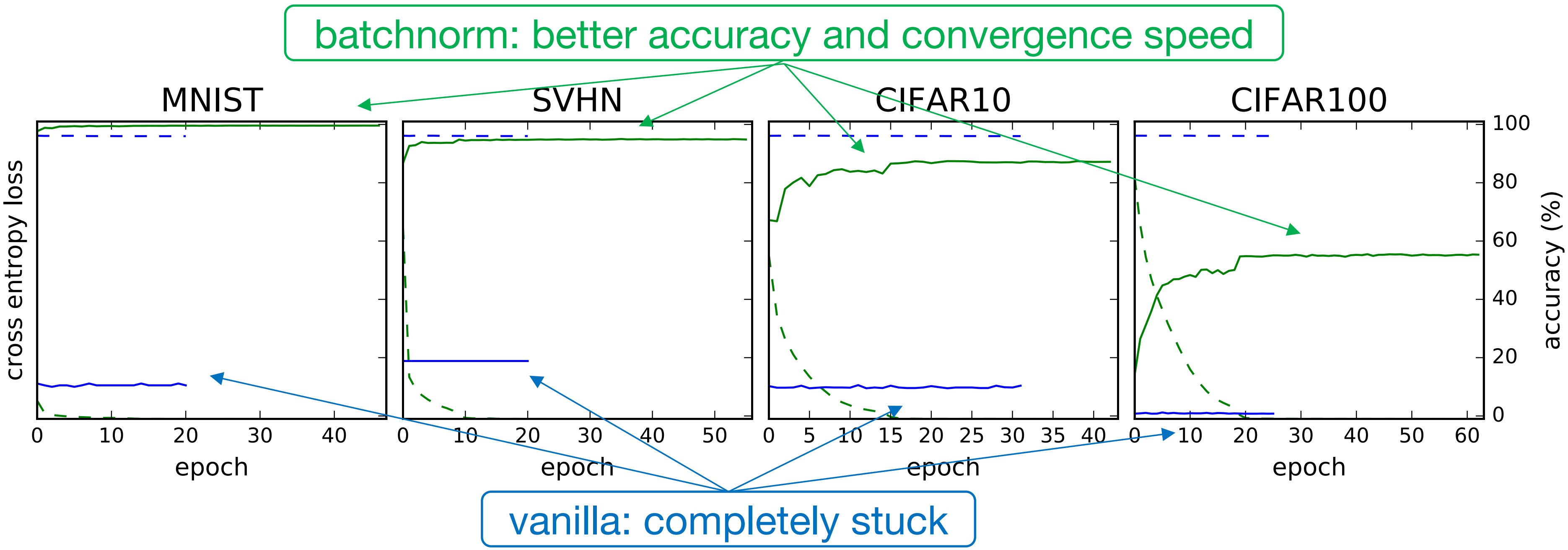
- Sigmoids (and $\tanh$ also) activation functions *saturate*
- This causes *vanishing gradient problems**



- Activations stuck in saturated regime
- More severe as network depth increases (multiplicative effect!)
- Batchnorm's scale and shift parameters push activations to non-saturated regime!

* Sepp Hochreiter: "Untersuchungen zu dynamischen neuronalen Netzen" (1991)

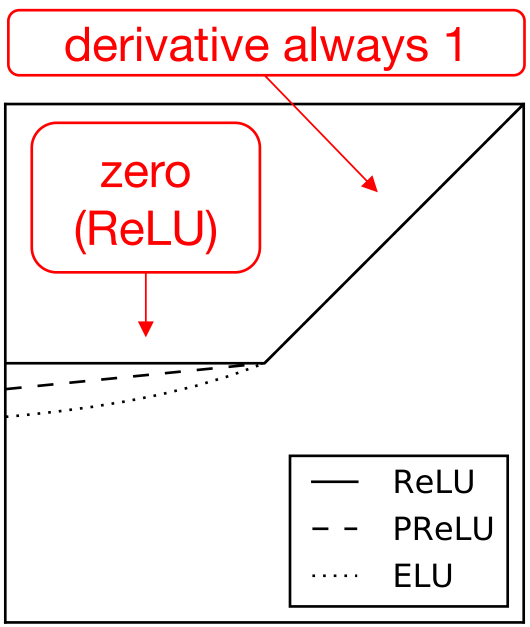
Batchnorm favors higher learning rates



- Back to ReLU baseline, increase (10x) initial learning rate to $\eta = 0.1$
- Batchnorm: **higher accuracy & convergence speed** than baseline
- Vanilla model is stuck at random guessing…
- Score: batchnorm 3 – vanilla 0
- What causes this behavior in the vanilla model?

A sidenote on rectified nonlinearities

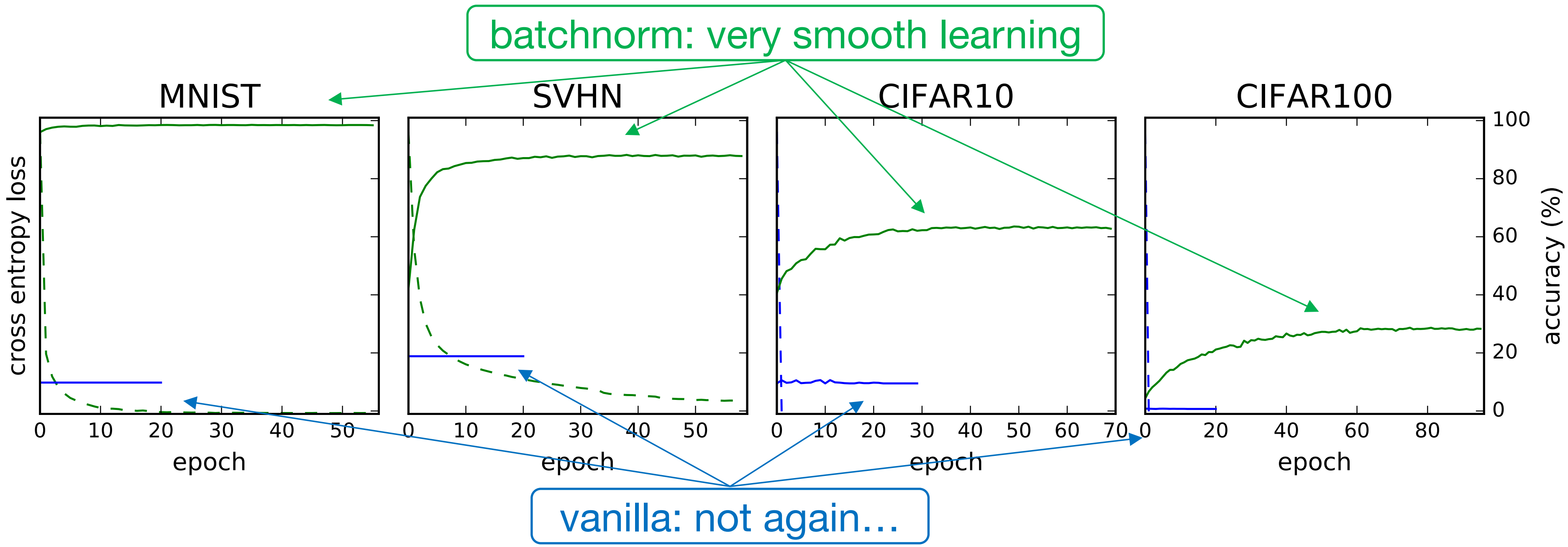
- ReLUs (also ELUs) are not bounded by saturation
- This causes *exploding gradient problems**



- Opposite of vanishing gradients!
- ReLUs cause so-called *dead neurons*
- Batchnorm scales and shifts activations and thus helps to prevents dead neurons
- Another approach: *gradient clipping*

* Razvan Pascanu et. al: “On the Difficulty of Training Recurrent Neural Networks” (2013)

Batchnorm handles arbitrary inits



- Back to ReLU baseline, initialize weights with $\mathcal{N}(\mu = 0, \sigma = 1)$
- Batchnorm: smooth learning but **worse accuracy** than baseline init
- Vanilla model is stuck again…
- Score: batchnorm 4 – vanilla 0
- What causes this behavior in the vanilla model?

A sidenote on weight initialization

Symmetry-breaking and *variance-preserving* inits crucial!

Batchnorm forward pass:

$$\text{batchnorm}\big((\alpha \boldsymbol{w})^T \boldsymbol{x} + \boldsymbol{b}\big) = \text{batchnorm}(\boldsymbol{w}^T \boldsymbol{x})$$

α has no effect

Batchnorm backward pass:

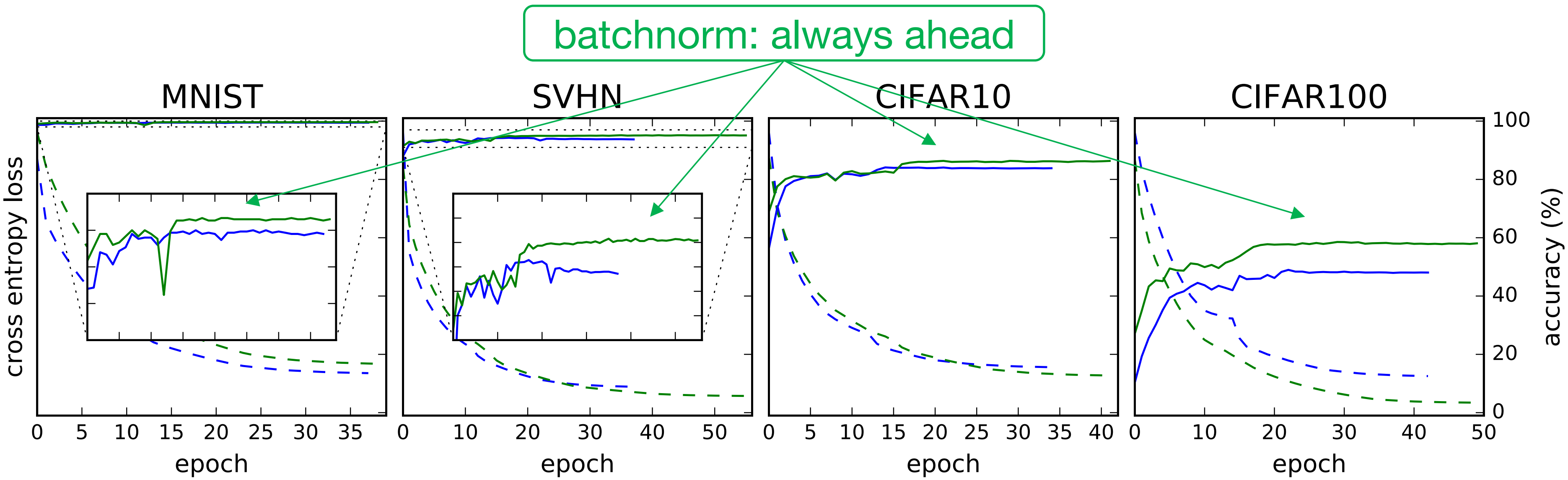
$$\frac{\partial\, \text{batchnorm}\big((\alpha \boldsymbol{w}^T \boldsymbol{x})\big)}{\partial\, \boldsymbol{x}} = \frac{\partial\, \text{batchnorm}(\boldsymbol{w}^T \boldsymbol{x})}{\partial\, \boldsymbol{x}}$$

α has no effect

$$\frac{\partial\, \text{batchnorm}\big((\alpha \boldsymbol{w}^T \boldsymbol{x})\big)}{\partial\, \alpha\boldsymbol{w}} = \frac{1}{\alpha} \cdot \frac{\partial\, \text{batchnorm}(\boldsymbol{w}^T \boldsymbol{x})}{\partial\, \boldsymbol{w}}$$

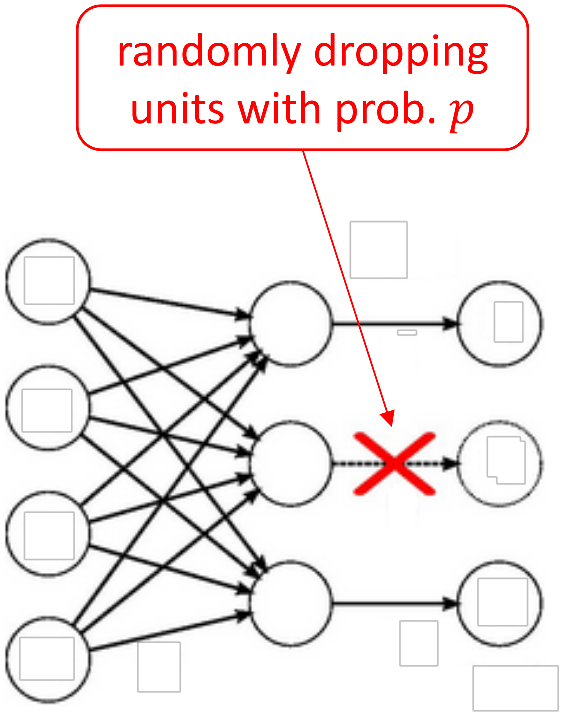
larger weights
→ smaller gradients
and vice versa!

Batchnorm likes some weight decay



- Back to baseline, add L_2 weight decay with strength $\lambda = 0.0005$
- Both models achieve **higher accuracies** than baseline models
- Batchnorm model consistently ahead of vanilla counterpart
- Score: batchnorm 5 – vanilla 0

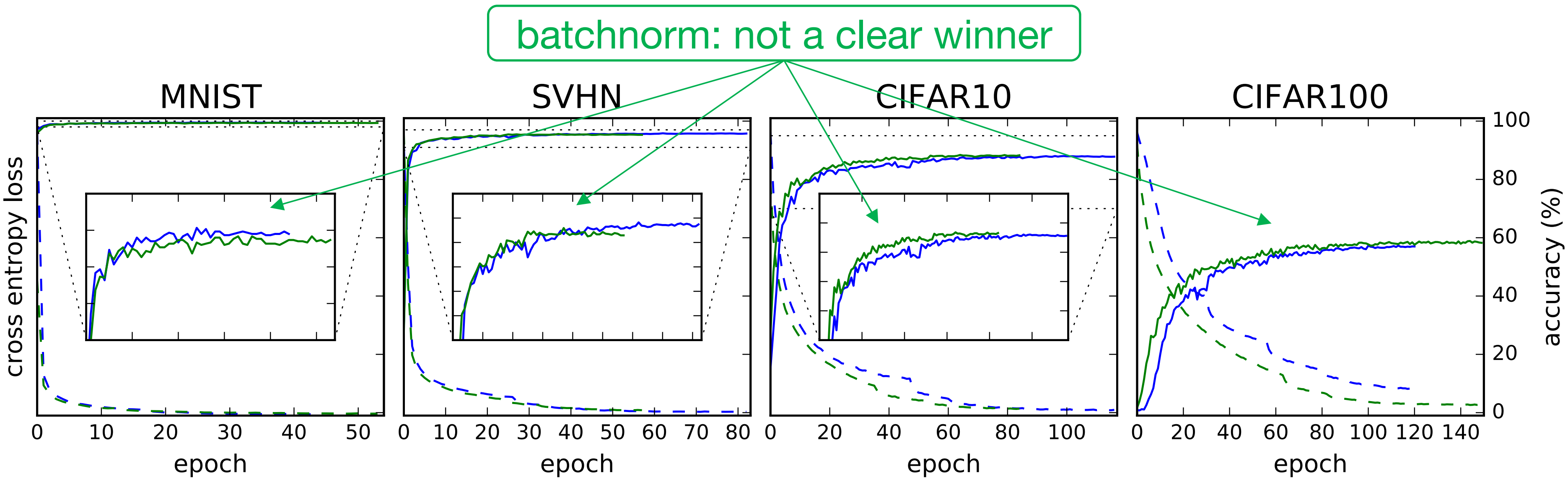
A quick intro to dropout *



- Prevents co-adaptation of features
- Can be seen as ensemble learning
- We do **not** drop units at test time!

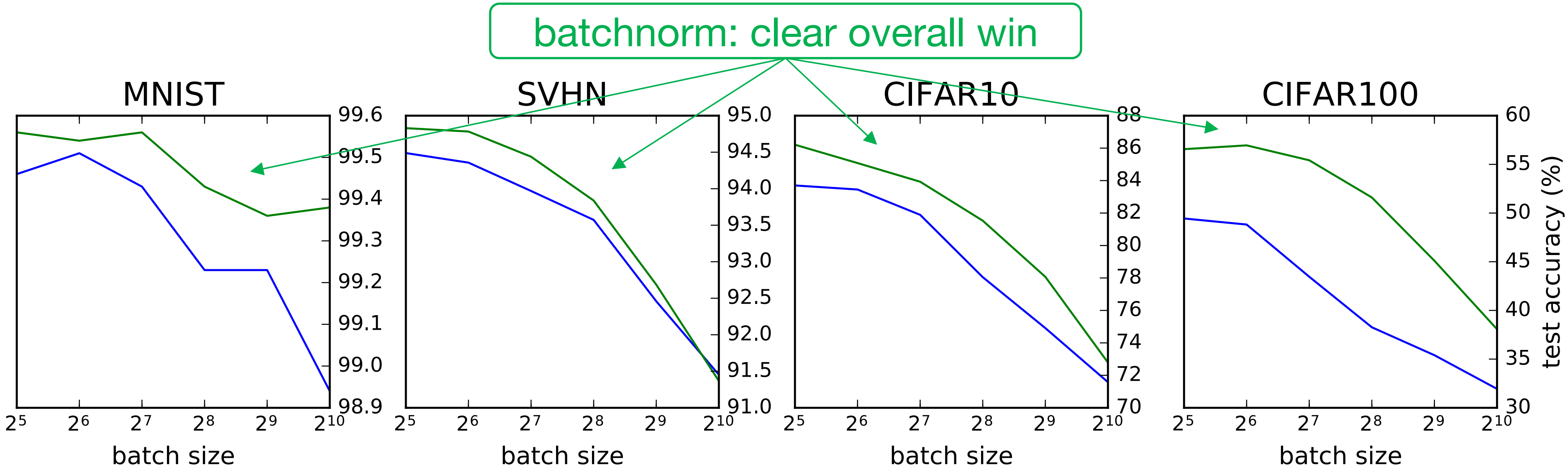
* Nitish Srivastava et. al: “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” (2014)

Batchnorm suffers under dropout



- Back to baseline, add dropout with keep probability $p = 0.5$
- Both models converge **much slower** than before, **best accuracies!**
- Batchnorm not ahead of vanilla counterpart
- Score: batchnorm 5 – vanilla 0 (we'll call this a draw!)
- Why? Batch statistics might be of lower value with dropped activations

Batchnorm can handle all batch sizes



- Back to baseline, we select batch sizes as in: $\mathcal{B} = \{2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}\}$
- Batchnorm ahead independent of batch size
- General trend: **smaller batch** sizes work better (for our models)
- Ultimately, it is a time/memory tradeoff!
- Score: batchnorm 6 – vanilla 0

Batchnorm time overhead noticeable

vanilla — batchnorm

Batch size	Mean overhead	MNIST V	MNIST B	SVHN V	SVHN B	CIFAR10 V	CIFAR10 B	CIFAR100 V	CIFAR100 B
32	19.7%	141	162	166	201	112	137	113	137
64	19.0%	103	122	128	153	87	104	87	104
128	16.4%	86	101	107	124	73	85	73	85
256	15.5%	77	89	95	109	65	75	65	75
512	15.3%	72	83	89	102	60	69	60	70
1024	15.2%	69	80	86	99	58	66	58	66

Table B.4: CNN average time per epoch for different batch sizes in seconds.

- Batchnorm scales nicely with increasing batch sizes
- Score: batchnorm 6 – vanilla 1 (we have to admit defeat here)
- **Your mileage may vary! ***

* Using CUDA 7.5 & cuDNN v4. Logging introduced overhead (but same for vanilla & batchnorm on absolute scale)

Conclusion: the effect of batchnorm

- What general effects does batchnorm have on…
  - …classification accuracy? Higher accuracies!
  - …convergence speed? Slightly faster! *

- What specific effects does batchnorm have on...
  - …activation functions? Can handle them all!
  - …learning rates? Higher learning rates favorable.
  - …weight initialization? Variance-preserving still best.
  - …regularization methods? Weight decay helps, not dropout. *
  - …batch sizes? Can handle all of them, smaller better. *
  - …wall time? Noticeable overhead, but worth it. *

* Very much in general. There are a lot of ”but if…”’s

Conclusion: why does batchnorm work?

- Examples are normalized over different batches which leads to a regularizing effect
- Vanishing and exploding gradients can be avoided due to scale and shift parameters
- More stable gradient propagation in general *
- Layers can “focus” on the predictive relationship, rather than adapting to shift in layer distributions

* Did not have time to show this, please refer to the thesis for an analysis

Key takeaway

Use batchnorm if your model has convergence problems!

Questions?

Backup slides

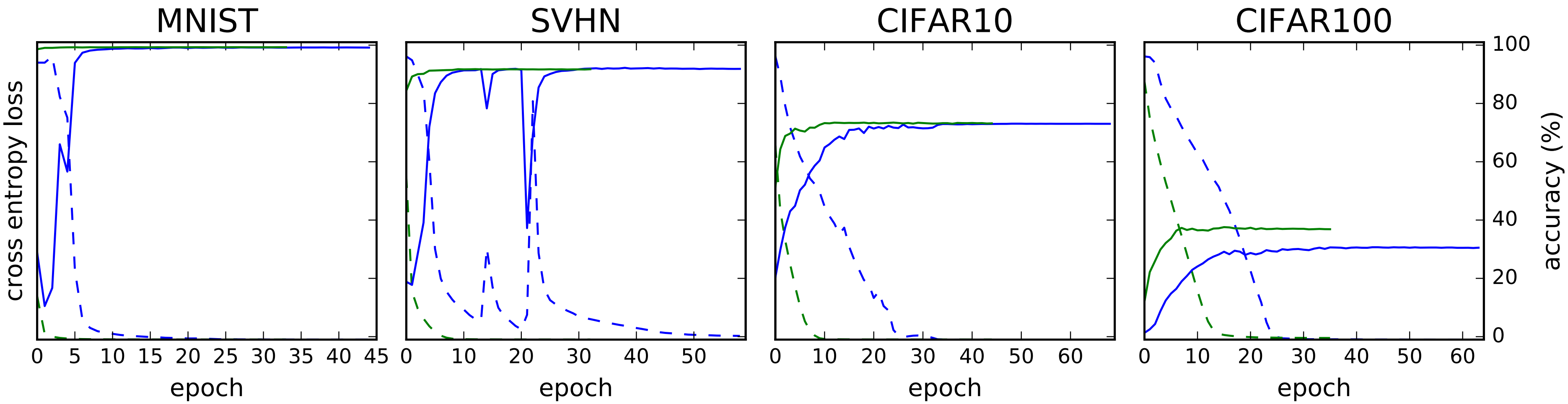
Did not have time to cover this

Training strategy for the experiments

- Global feature standardization
- Shuffle training set, 10% validation split
- Cross entropy loss
- Mini-batch SGD with batch size $\mathcal{B} = 64$
  - **Nesterov momentum** * with $\mu = 0.9$
- Initial learning rate of $\eta = 0.01$
- Adaptive learning rate decay with $k = 0.5$
  - After 5 epochs without improvement
- "Early" stopping after 20 epochs without improvement

* Ilya Sutskever et. al "On the Importance of Initialization and Momentum in Deep Learning" (2013)

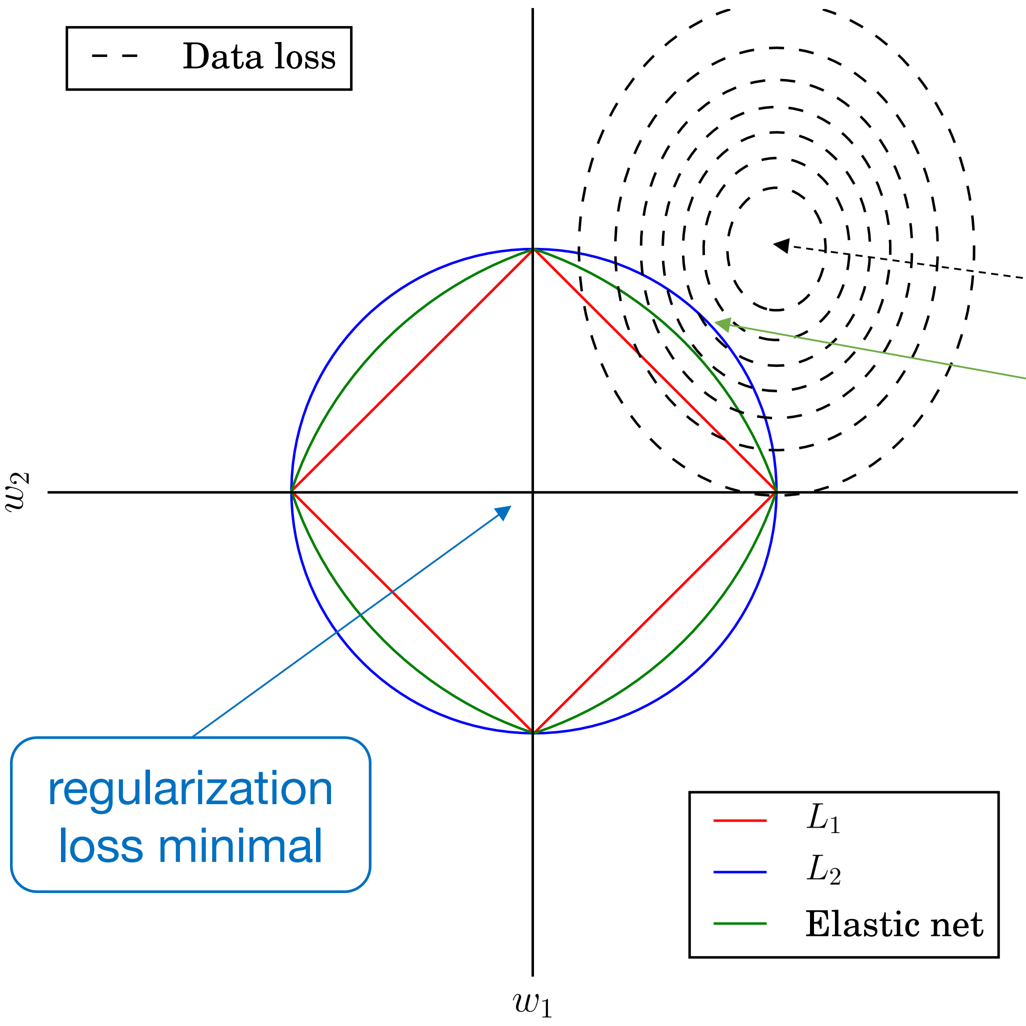
Funky convergence problems



An intro to L_2 weight decay

add L_2 norm to loss

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}, \boldsymbol{w}) = \frac{1}{\mathcal{B}} \sum_{i=1}^{\mathcal{B}} \mathcal{L}_i(\hat{y}_i, y_i) + \frac{\lambda}{2} ||w||_2^2$$



- Gaussian prior on weights
- Perform MAP instead of MLE
- Independent of batchnorm?

SGD: what could possibly go wrong?

note: effects up to order l!

We have the following simple model:

$$f(x, \boldsymbol{w}) = x \cdot w_1 \cdot w_2 \cdot \ldots \cdot w_l = \hat{y}$$

Assume $\frac{\partial \mathcal{L}}{\partial \hat{y}} = 1$ so we wish to decrease $\hat{y}$ slighly.

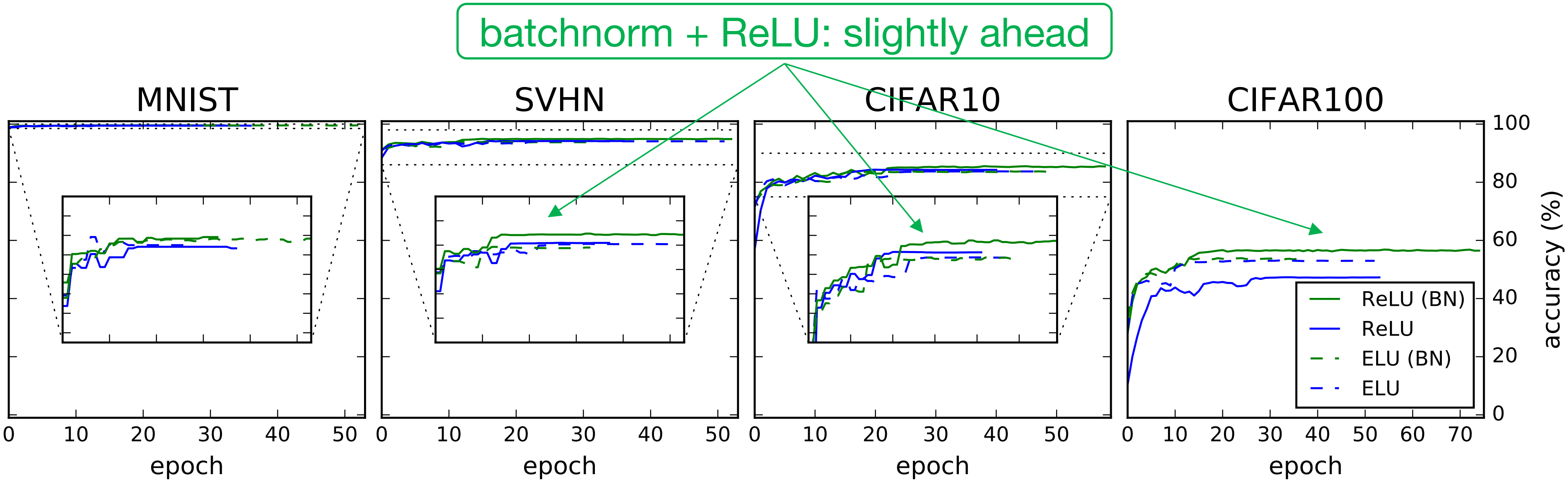
After gradient descent update the network computes:

$$f(x, \boldsymbol{w}) = x \cdot \left(w_1 - \eta \frac{\partial \mathcal{L}}{\partial w_1}\right) \cdot \left(w_2 - \eta \frac{\partial \mathcal{L}}{\partial w_2}\right) \cdot \ldots \cdot \left(w_l - \eta \frac{\partial \mathcal{L}}{\partial w_l}\right) = \hat{y}$$

adaptive optimization can help!

how can wechoose the learning rate?

Batchnorm + ReLU outperforms ELU*



- Train both models with ELU nonlinearities
- Paper claims that vanilla ELUs outperform ReLU models with batchnorm
- We found that this is **not** the case (at least for our setup)
- Theoretical properties of ELUs are questionable

* D. Clevert et. al: "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)" (2015)

Conclusion: practical recommendations

- Initial learning rate & its decay are crucial
  - If you optimize one thing, optimize those!
- Use ReLU (give parametric & leaky ReLU a try)
- Use dropout!
- Try residual connections and inception
- Use batchnorm if convergence seems problematic