

Lecture 4

Going deeper into Deep Learning

Goncharenko Vladislav

Moscow, 2021

1. Previous lecture recap: backpropagation, activations, intuition.
2. Optimizers.
3. Data normalization.
4. Regularization.
5. PyTorch practice.
6. Q & A.

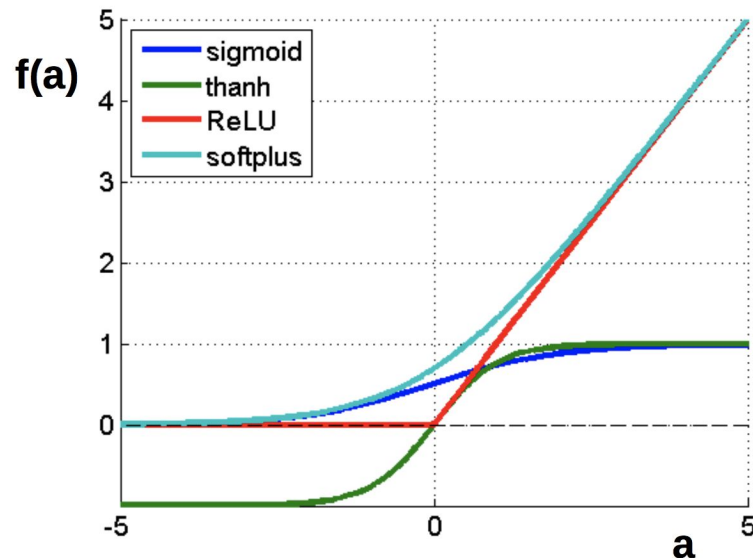
Once more: nonlinearities

$$f(a) = \frac{1}{1 + e^a}$$

$$f(a) = \tanh(a)$$

$$f(a) = \max(0, a)$$

$$f(a) = \log(1 + e^a)$$

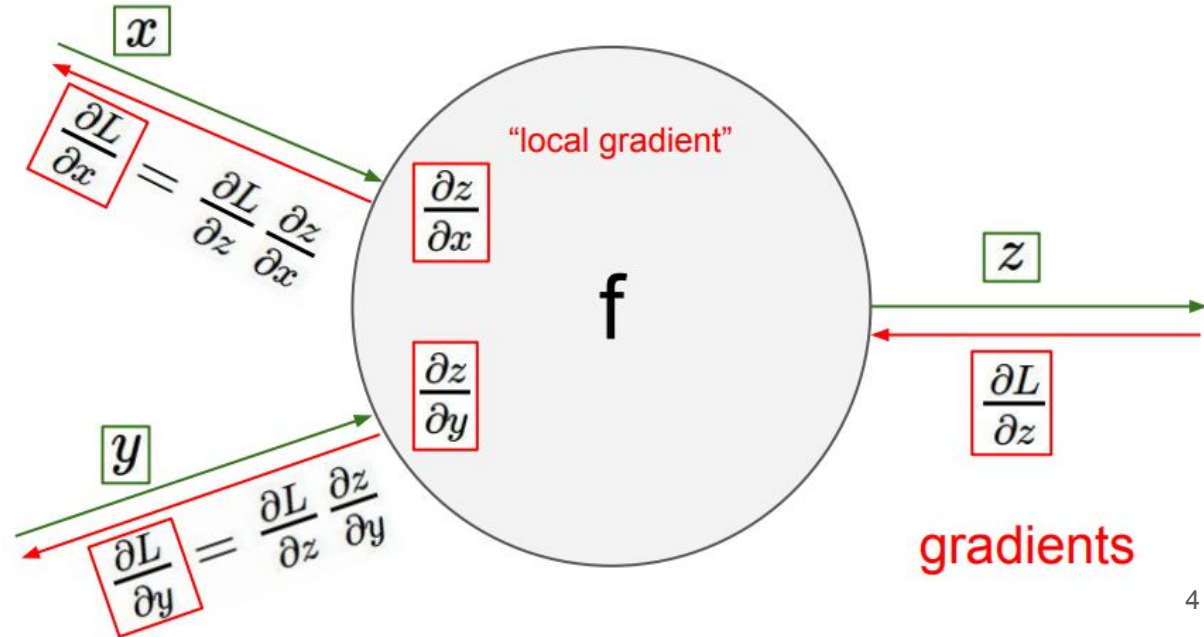


Backpropagation and chain rule

Chain rule is just simple math:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$$

Backprop is just way to use it in NN training.



- Layers
 - a. Dense layer (*done*)
 - b. Convolutional layer (*next lecture*)
 - c. Pooling layer (*next lecture*)
 - d. Dropout layer (*today*)
 - e. Batchnorm layer (batch normalization) (*today*)
 - f. Embeddings (aka word2vec, GloVe) (*last lecture*)
 - g. Recurrent layers (*last lecture*)

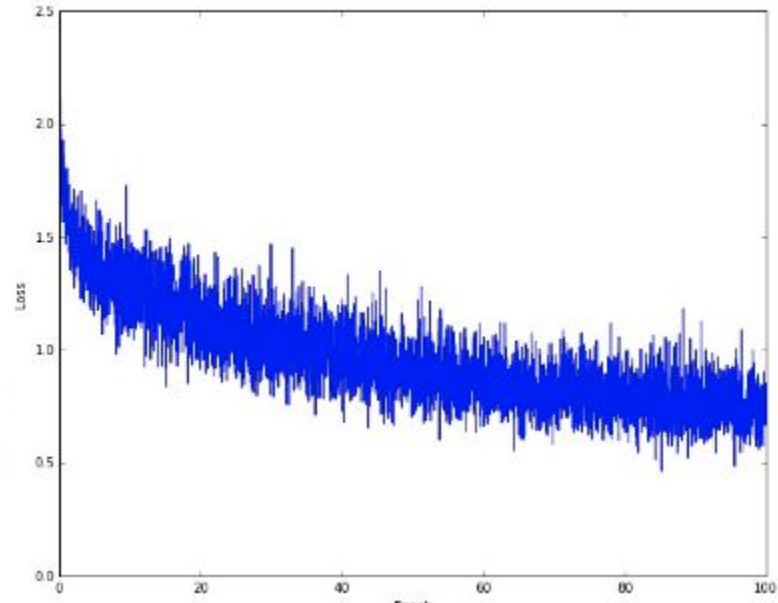
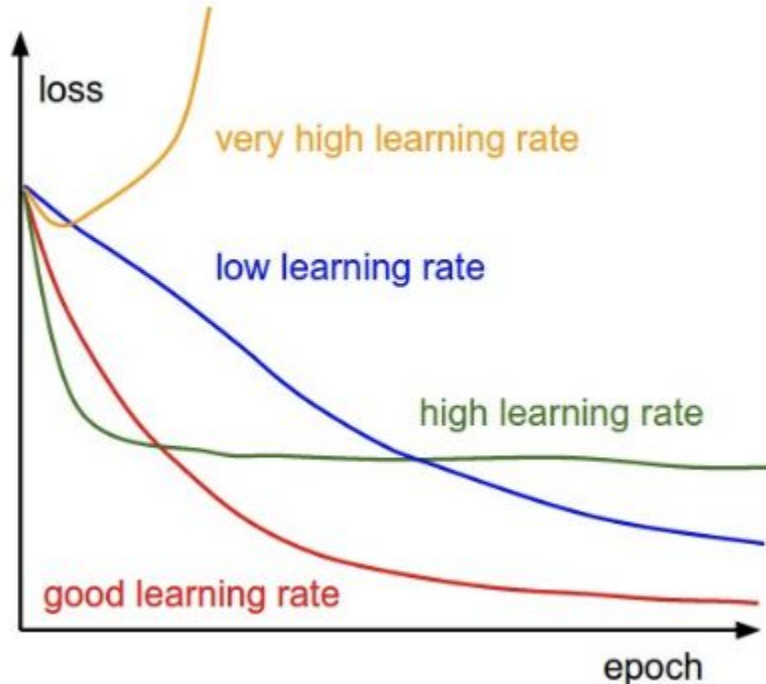
Different layers

- Layers
 - a. Dense layer (*done*)
 - b. Convolutional layer (*next lecture*)
 - c. Pooling layer (*next lecture*)
 - d. Dropout layer (*today*)
 - e. Batchnorm layer (batch normalization) (*today*)
 - f. Embeddings (aka word2vec, GloVe) (*last lecture*)
 - g. Recurrent layers (*last lecture*)

Optimizers

Stochastic gradient descent is used to optimize NN parameters.

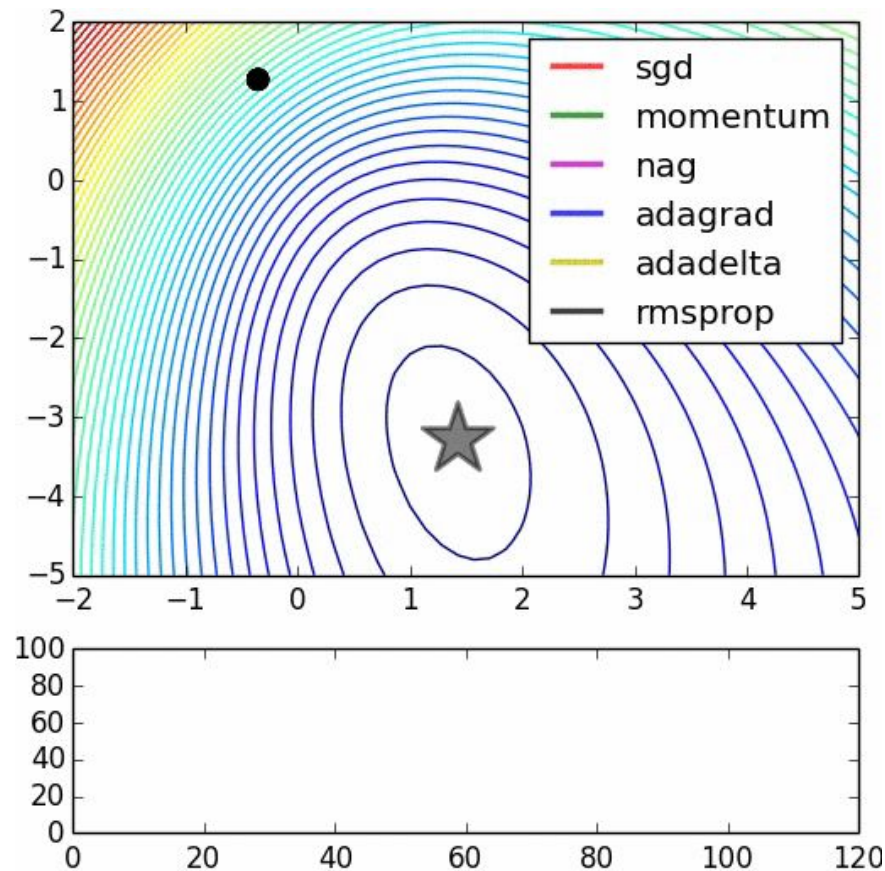
$$x_{t+1} = x_t - \text{learning rate} \cdot dx$$



Optimizers

There are much more optimizers:

- Momentum
- Adagrad
- Adadelata
- RMSprop
- Adam
- ...
- even other NNs

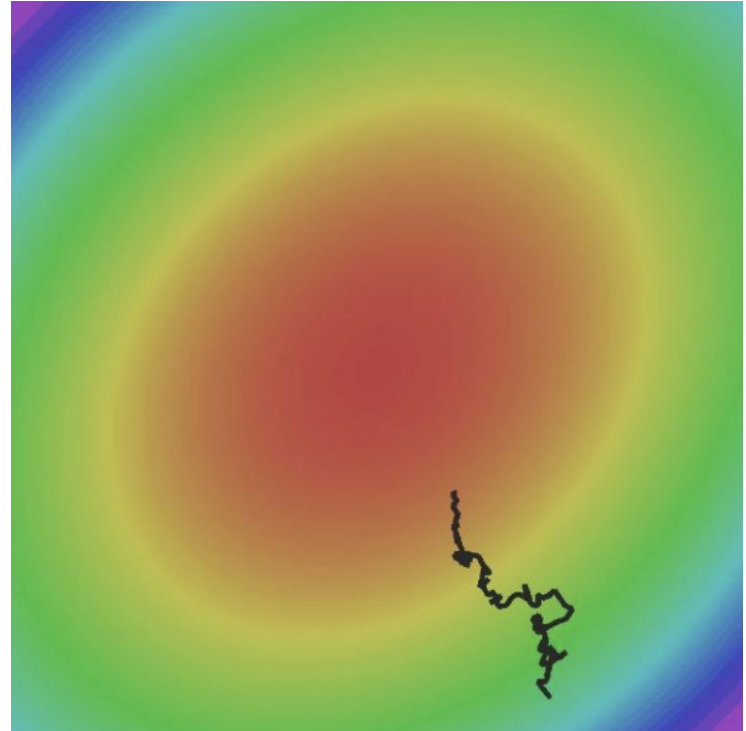


Optimization: SGD

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

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

Averaging over minibatches ---> noisy gradient



First idea: momentum

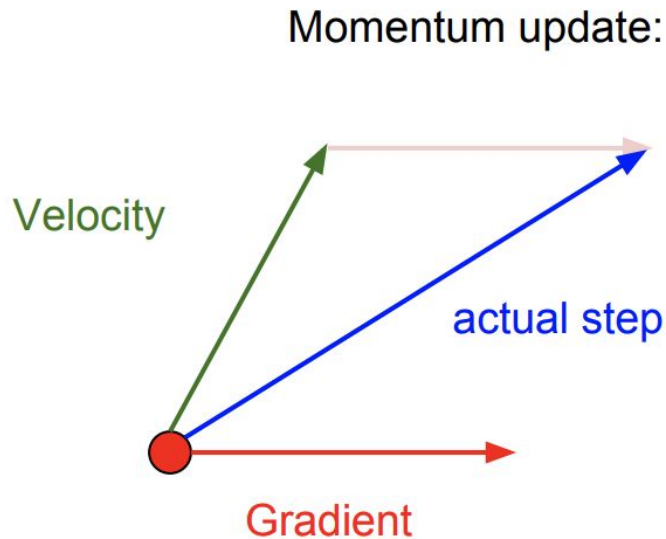
Simple SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

SGD with momentum

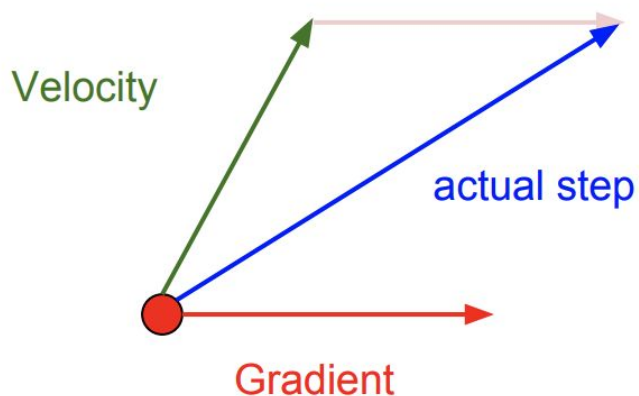
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$



Nesterov momentum

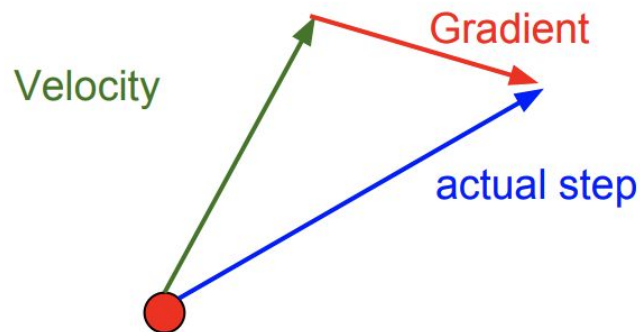
Momentum update:



$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

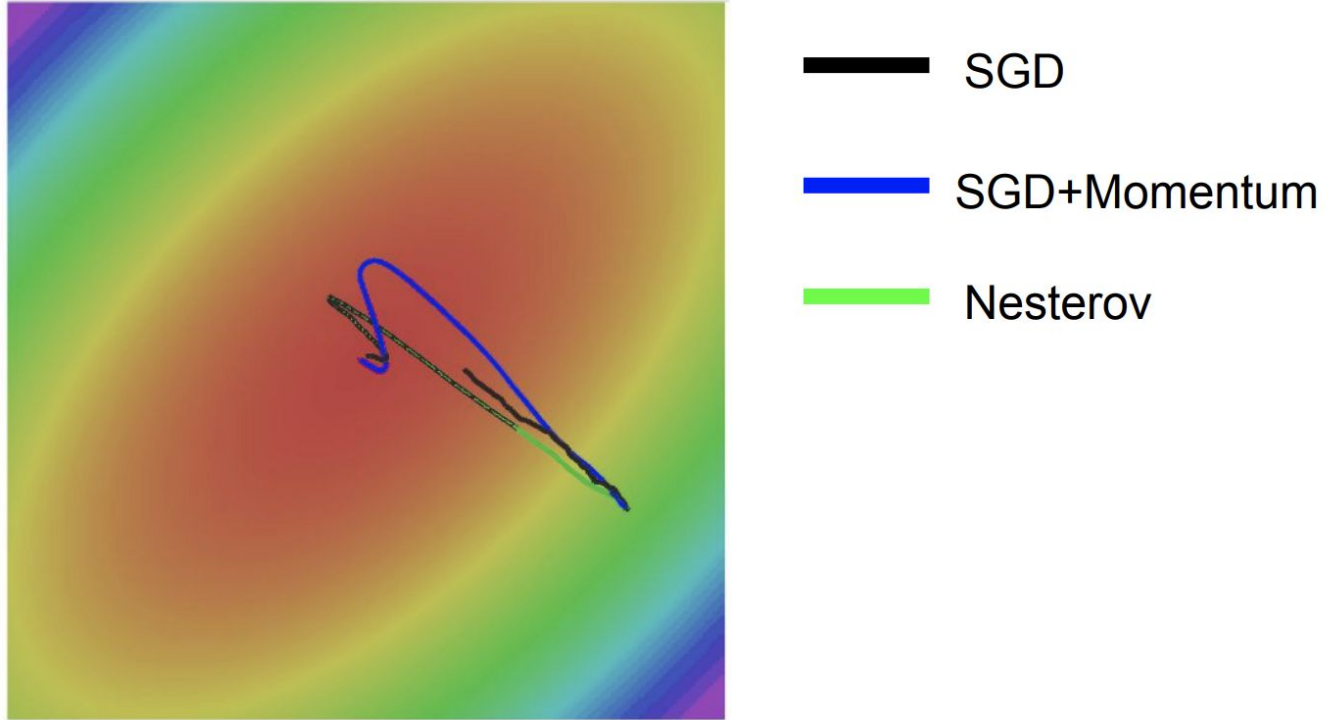
Nesterov Momentum



$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

Comparing momentums



Second idea: different dimensions are different

Adagrad: SGD with cache

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Second idea: different dimensions are different

Adagrad: SGD with cache

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Problem: gradient fades with time

Second idea: different dimensions are different

Adagrad: SGD with cache

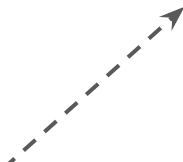
$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

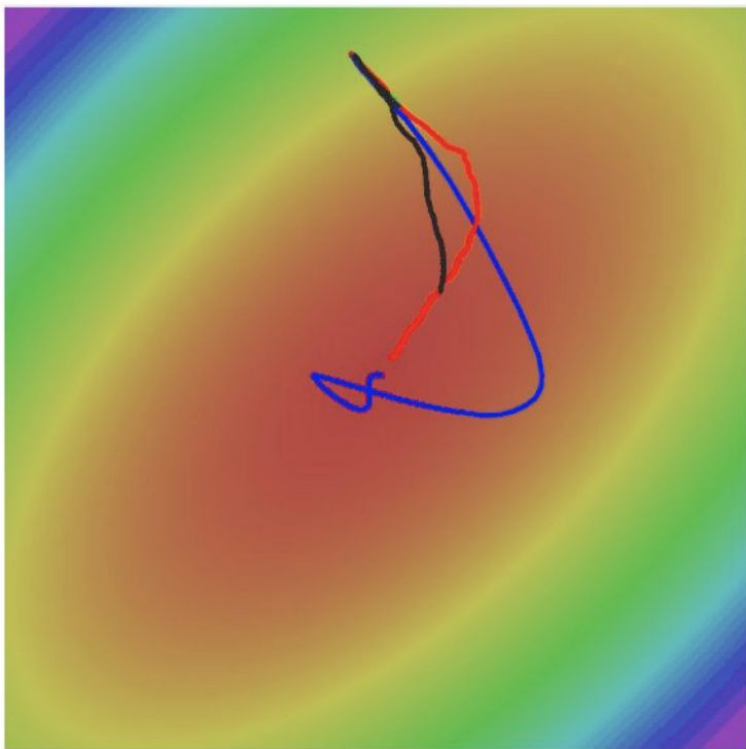
$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$



RMSProp: SGD with cache with exp. Smoothing

$$\text{cache}_{t+1} = \beta \text{cache}_t + (1 - \beta)(\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$




— SGD

— SGD+Momentum

— RMSProp

Let's combine the momentum idea and RMSProp normalization:

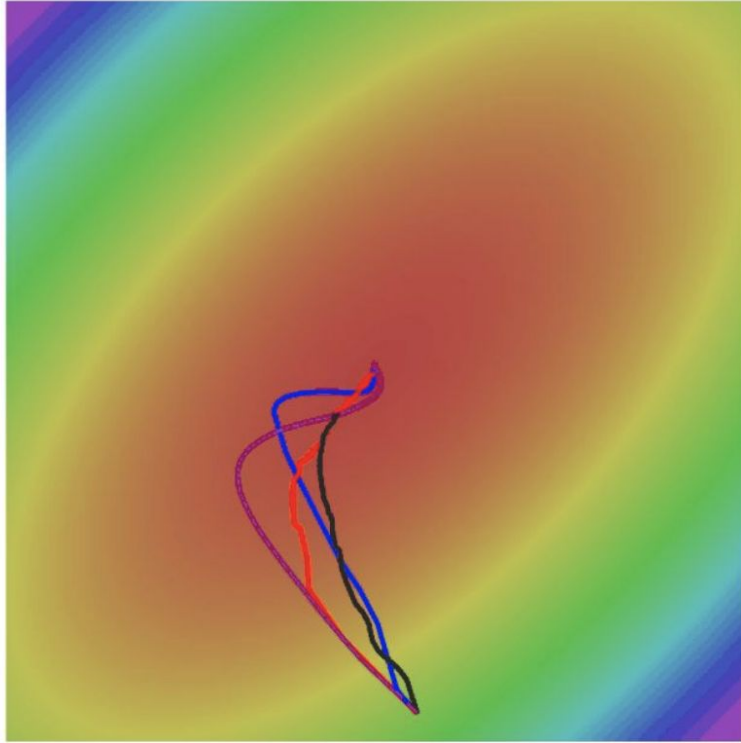
$$\begin{aligned}v_{t+1} &= \gamma v_t + (1 - \gamma) \nabla f(x_t) \\ \text{cache}_{t+1} &= \beta \text{cache}_t + (1 - \beta) (\nabla f(x_t))^2 \\ x_{t+1} &= x_t - \alpha \frac{v_{t+1}}{\text{cache}_{t+1} + \varepsilon}\end{aligned}$$

Let's combine the momentum idea and RMSProp normalization:

$$\begin{aligned}v_{t+1} &= \gamma v_t + (1 - \gamma) \nabla f(x_t) \\ \text{cache}_{t+1} &= \beta \text{cache}_t + (1 - \beta) (\nabla f(x_t))^2 \\ x_{t+1} &= x_t - \alpha \frac{v_{t+1}}{\text{cache}_{t+1} + \varepsilon}\end{aligned}$$

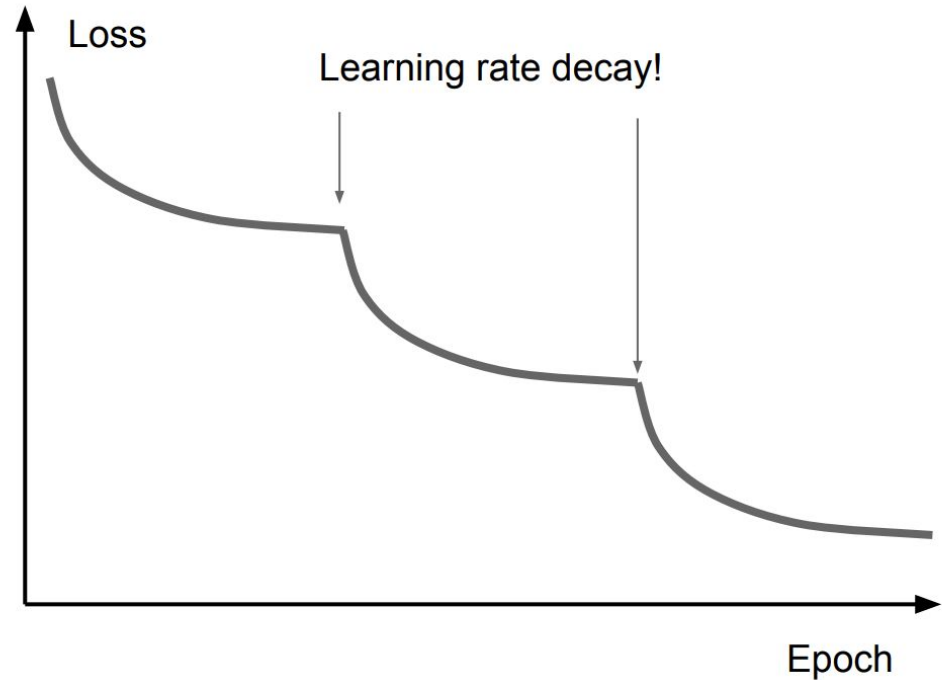
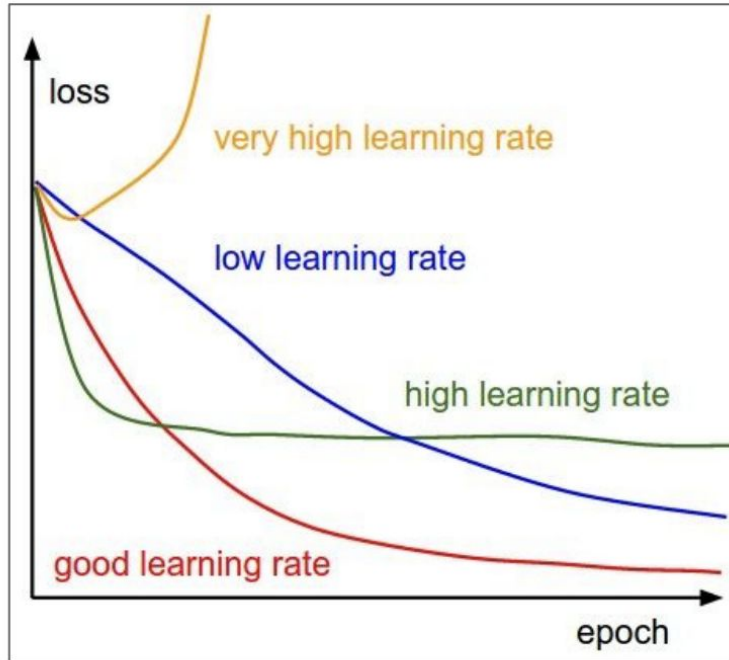
Actually, that's not quite Adam.

Comparing optimizers



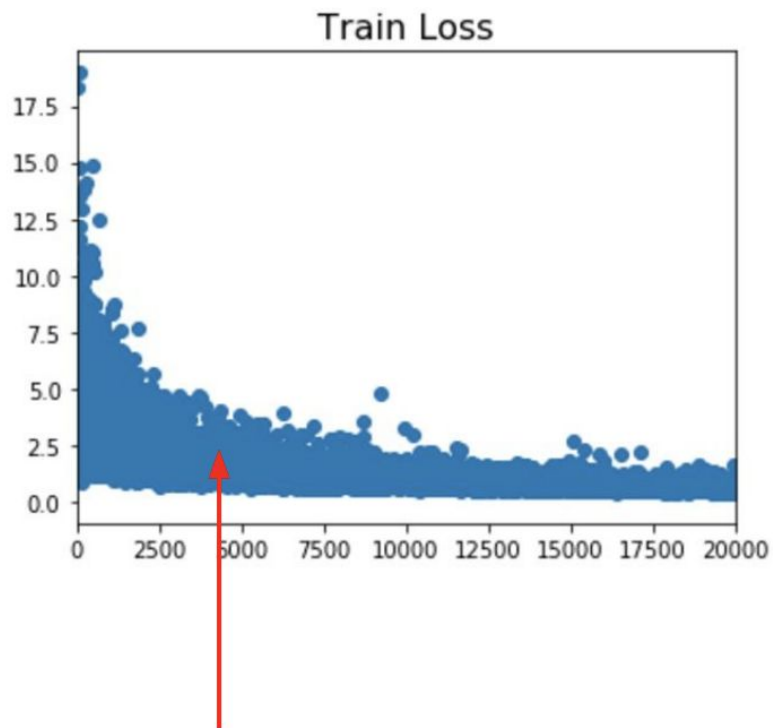
- SGD
- SGD+Momentum
- RMSProp
- Adam

Once more: learning rate

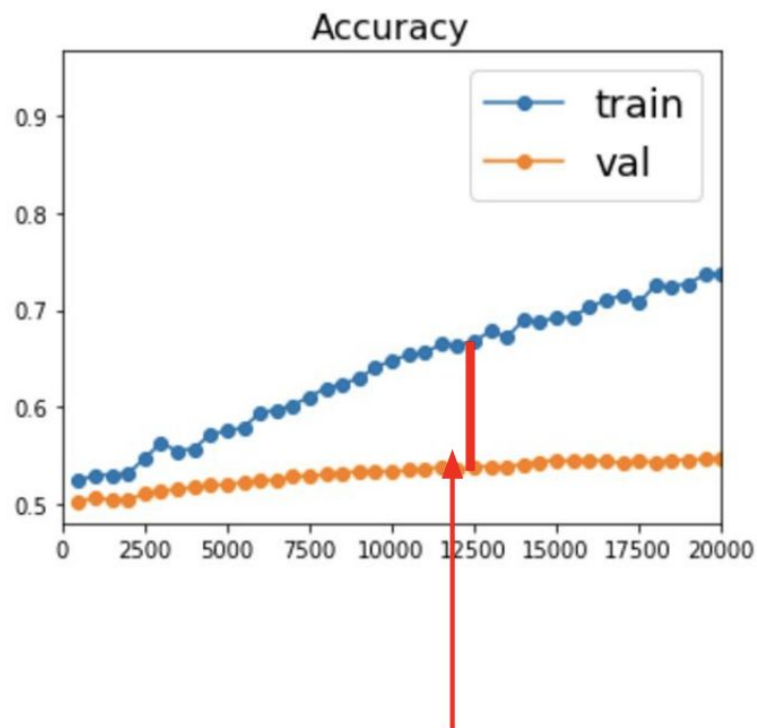


Sum up: optimization

- Adam is great basic choice
- Even for Adam/RMSProp learning rate matters
- Use learning rate decay
- Monitor your model quality

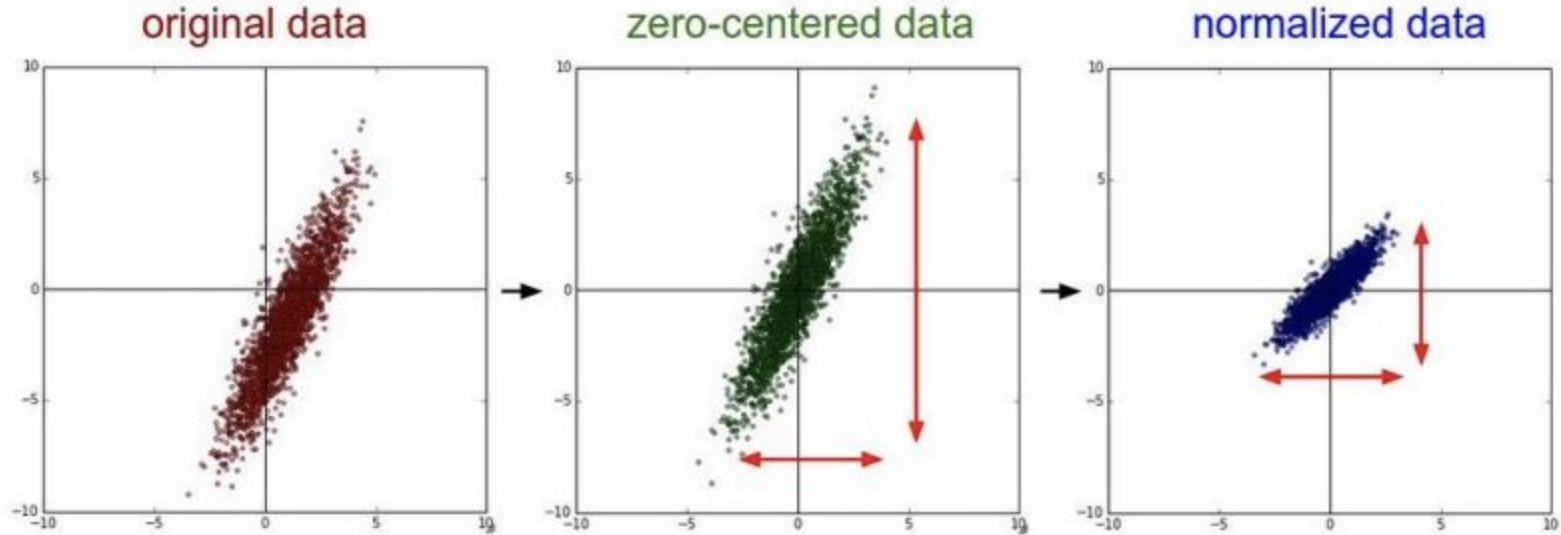


Better optimization algorithms
help reduce training loss



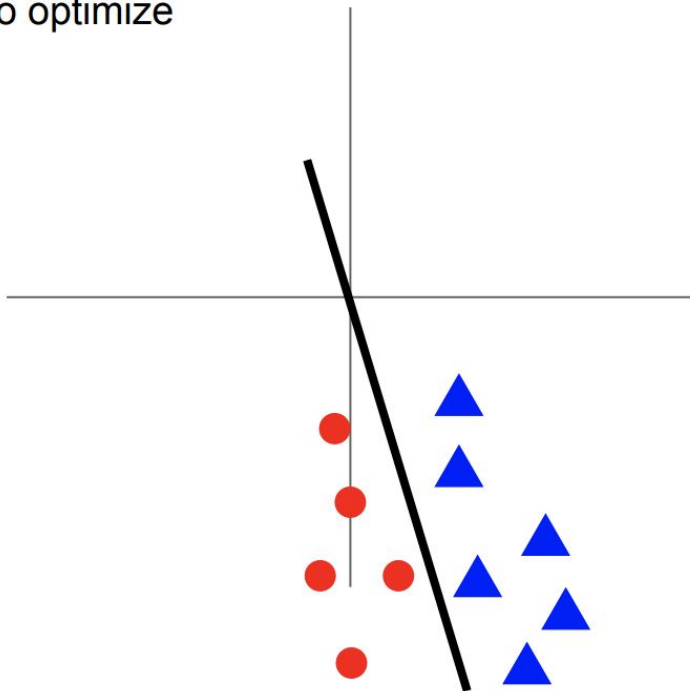
But we really care about error on new
data - how to reduce the gap?

Data normalization

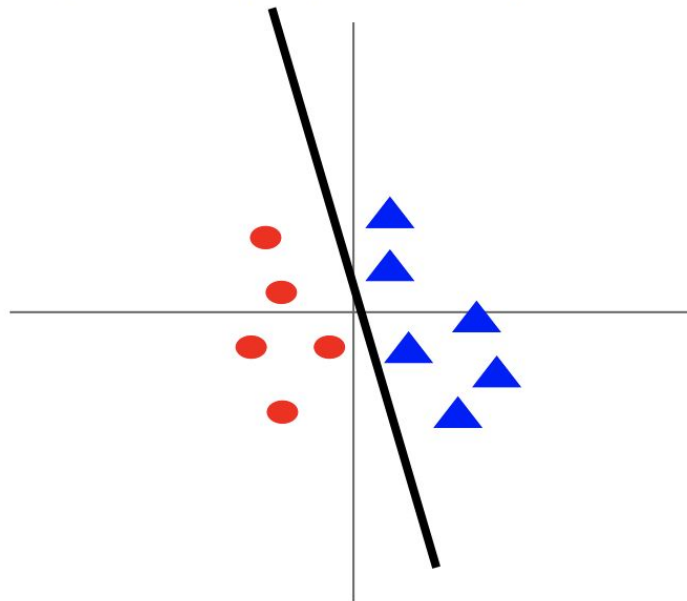


Data normalization

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Weights initialization

- Pitfall: all zero initialization.

Weights initialization

- Pitfall: all zero initialization.
- Small random numbers.

Weights initialization

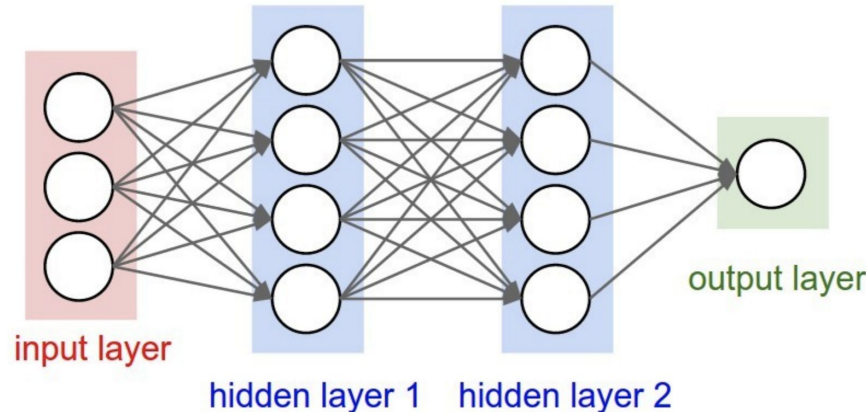
- Pitfall: all zero initialization.
- Small random numbers.
- Calibrated random numbers.

$$\begin{aligned} s &= \sum_i^n w_i x_i \\ \text{Var}(s) &= \text{Var}\left(\sum_i^n w_i x_i\right) \\ &= \sum_i^n \text{Var}(w_i x_i) \\ &= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \\ &= \sum_i^n \text{Var}(x_i) \text{Var}(w_i) \\ &= (n \text{Var}(w)) \text{Var}(x) \end{aligned}$$

Batch normalization

Problem:

- Consider a neuron in any layer beyond first
- At each iteration we tune it's weights towards better loss function
- But we also tune it's inputs. Some of them become larger, some – smaller
- Now the neuron needs to be re-tuned for it's new inputs



Batch normalization

TL; DR:

- It's usually a good idea to normalize linear model inputs
- (c) Every machine learning lecturer, ever

Batch normalization

- Normalize activation of a hidden layer
(zero mean unit variance)

$$h_i = \frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}$$

- Update μ_i, σ_i^2 with moving average while training

$$\mu_i := \alpha \cdot \text{mean}_{\text{batch}} + (1 - \alpha) \cdot \mu_i$$

$$\sigma_i^2 := \alpha \cdot \text{variance}_{\text{batch}} + (1 - \alpha) \cdot \sigma_i^2$$

Batch normalization

Original algorithm (2015)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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

Batch normalization

Original algorithm (2015)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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

What is this?

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

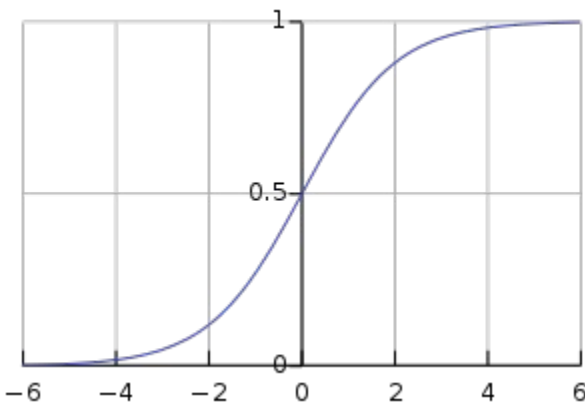
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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



Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

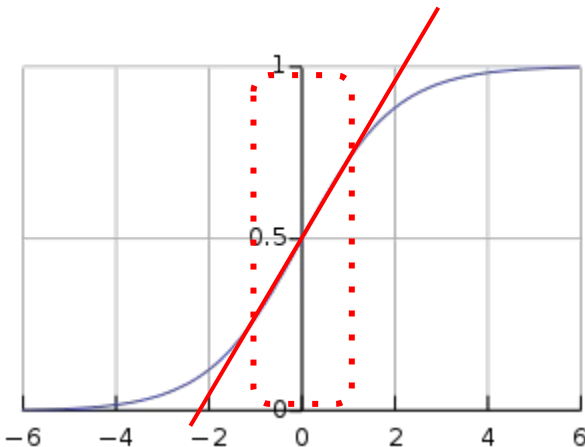
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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



Batch normalization

Original algorithm (2015)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

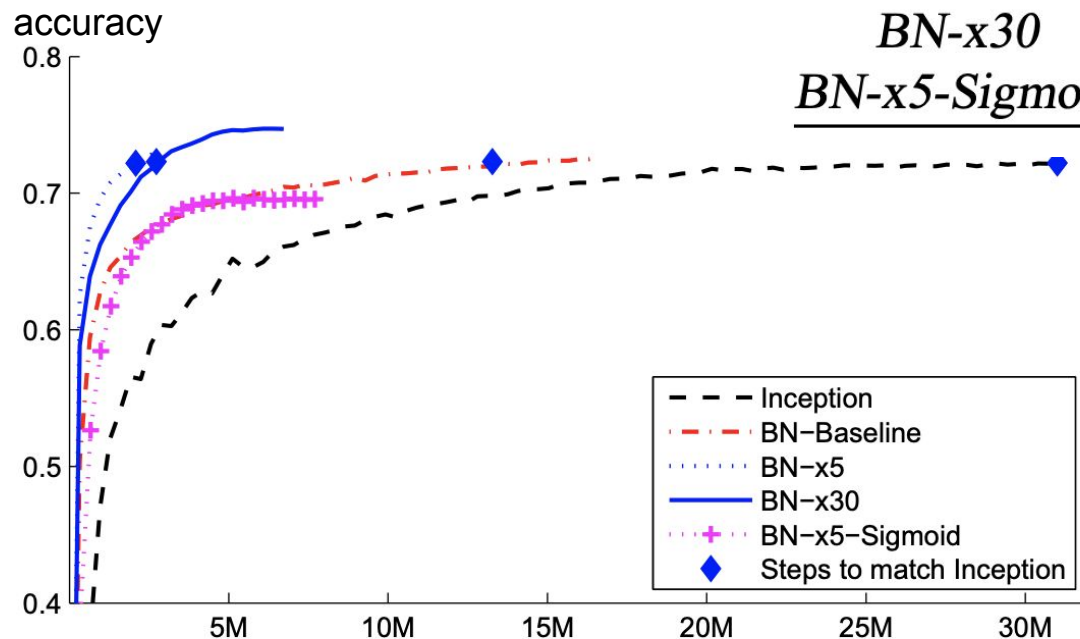
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

What is this?

This transformation should be able to represent the identity transform.

Batch normalization

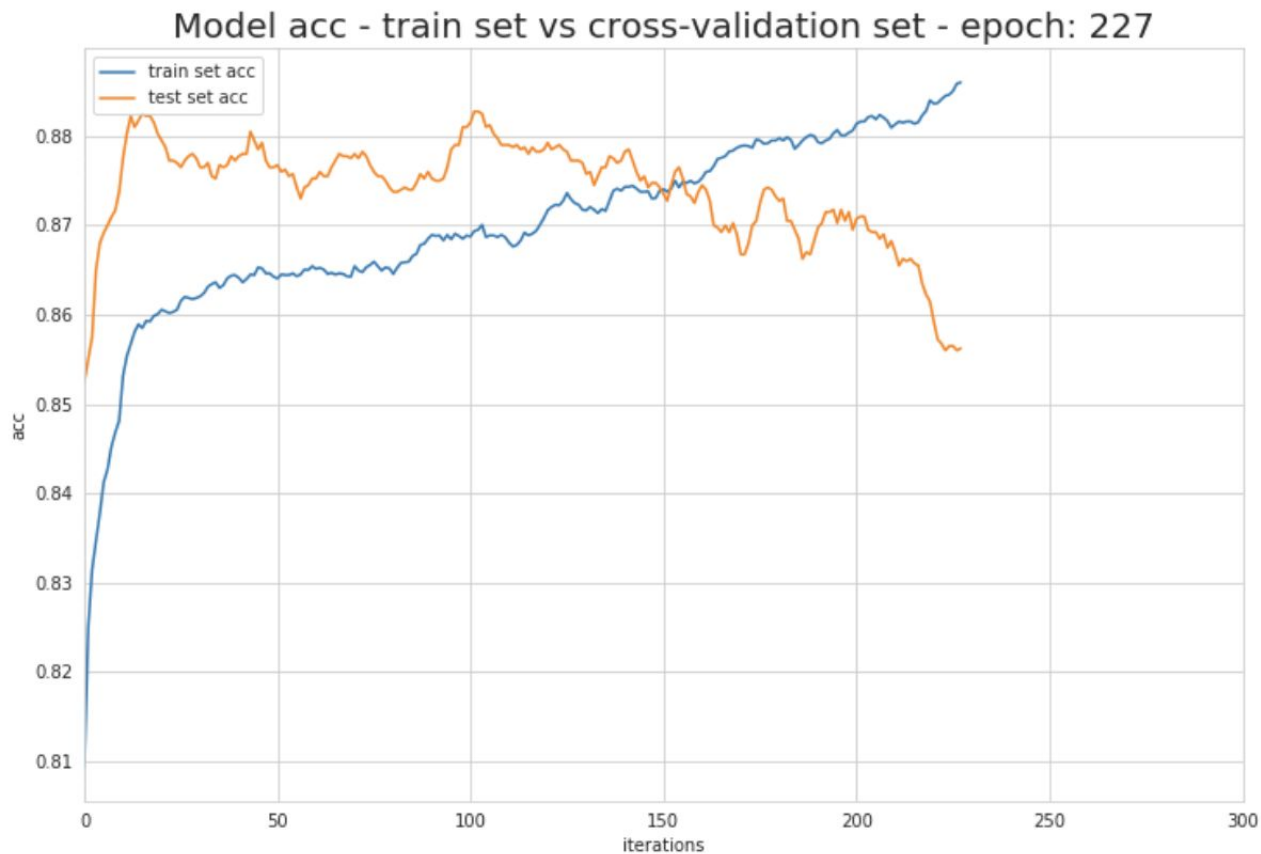
Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
<i>BN-Baseline</i>	$13.3 \cdot 10^6$	72.7%
<i>BN-x5</i>	$2.1 \cdot 10^6$	73.0%
<i>BN-x30</i>	$2.7 \cdot 10^6$	74.8%
<i>BN-x5-Sigmoid</i>		69.8%



source: <https://arxiv.org/pdf/1502.03167.pdf>

number of training steps

Problem: overfitting



$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

Adding some extra term to the loss function.

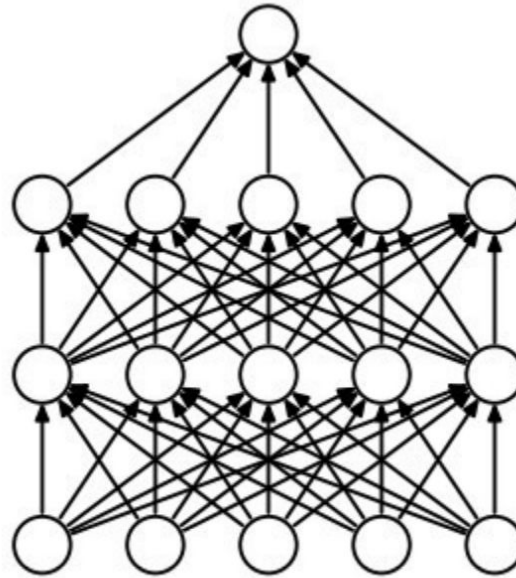
Common cases:

- L2 regularization: $R(W) = \|W\|_2^2$
- L1 regularization: $R(W) = \|W\|_1$
- Elastic Net (L1 + L2): $R(W) = \beta \|W\|_2^2 + \|W\|_1$

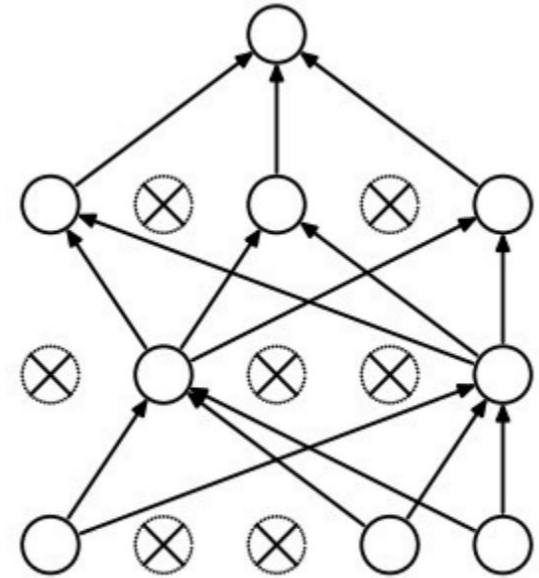
Regularization: Dropout

Some neurons are “dropped” during training.

Prevents overfitting.



(a) Standard Neural Net

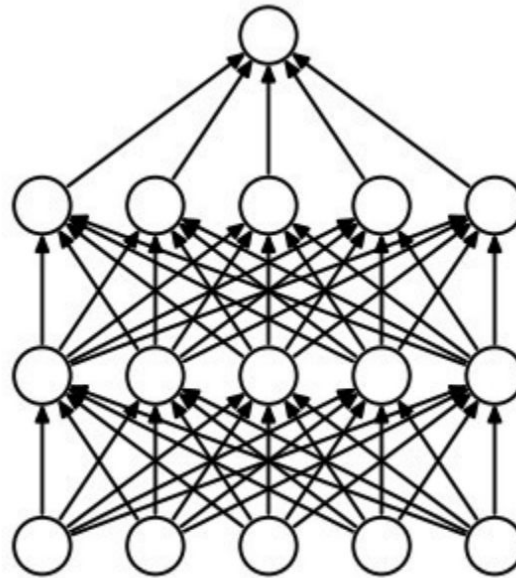


(b) After applying dropout.

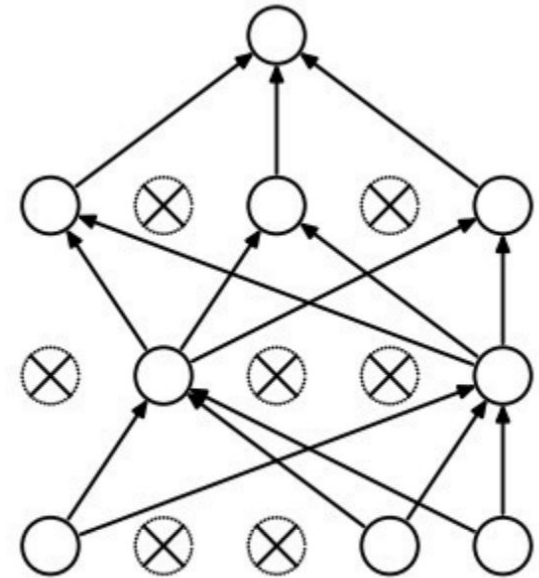
Regularization: Dropout

Some neurons are “dropped” during training.

Prevents overfitting.



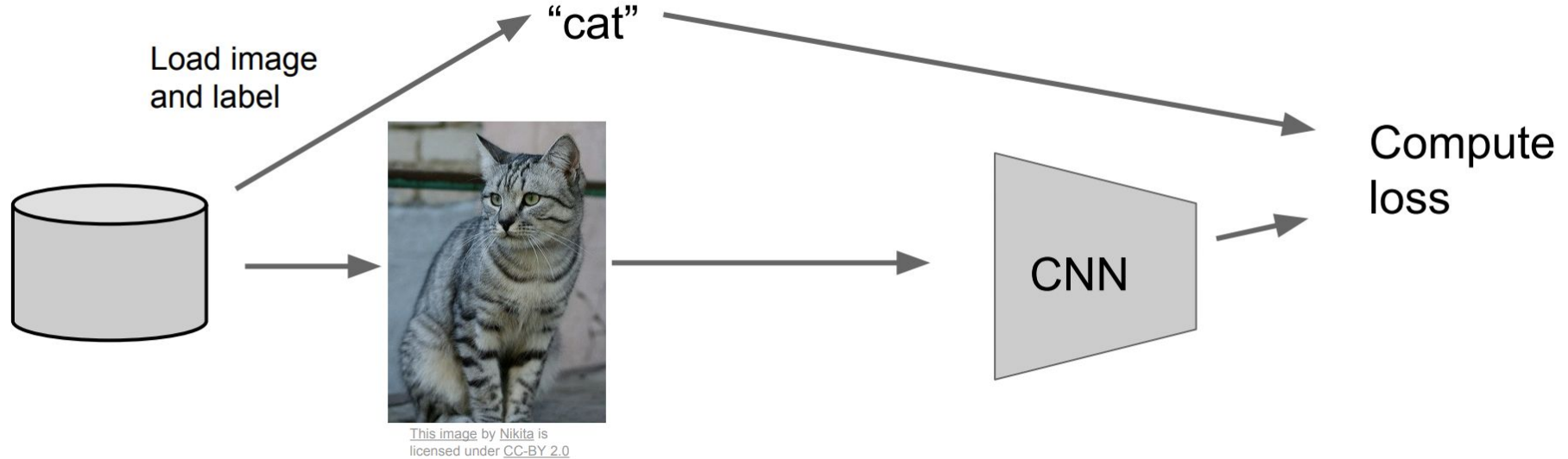
(a) Standard Neural Net



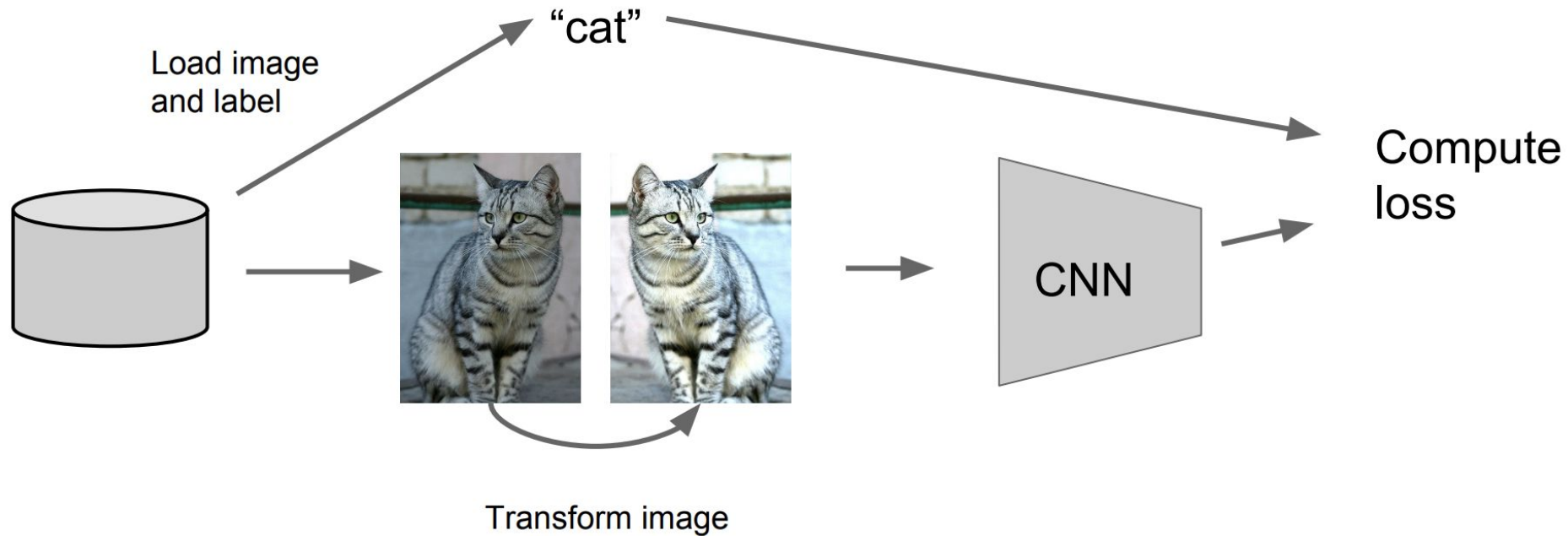
(b) After applying dropout.

Actually, on test case output should be normalized. See sources for more info.

Regularization: data augmentation



Regularization: data augmentation



Sum up: regularization

Regularization:

- Add some weight constraints
- Add some random noise during train and marginalize it during test
- Add some prior information in appropriate form

That's all. Feel free to ask any questions.