Training (convolutional) neural networks for classification

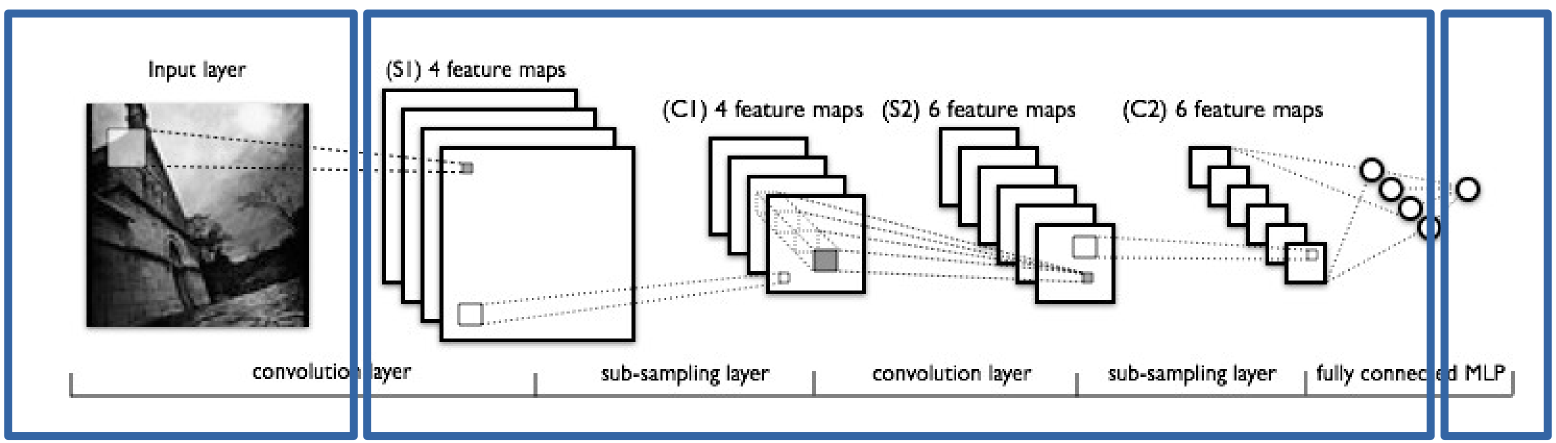
Mitko Veta
IMAG/e, Eindhoven University of Technology

Outline

- Gradient descent
- Loss function
- Backpropagation
  - The modular way
- Implementing a layer
  - Example: linear layer in caffe
- Regularization
- Initialization
- Some practical considerations

Training neural networks

- Find "good" values for $\boldsymbol{\theta}$



$\mathbf{x}^{(d)}$ $\qquad$ $\mathbf{f}(\mathbf{x},\boldsymbol{\theta})$ $\qquad$ $\mathbf{f}(\mathbf{x}^{(d)},\boldsymbol{\theta})$

$y^{(d)}$ $\qquad$ $y^{(d)}$

Image from: http://deeplearning.net/

Training neural networks

- The loss informs us how “good” the parameters are

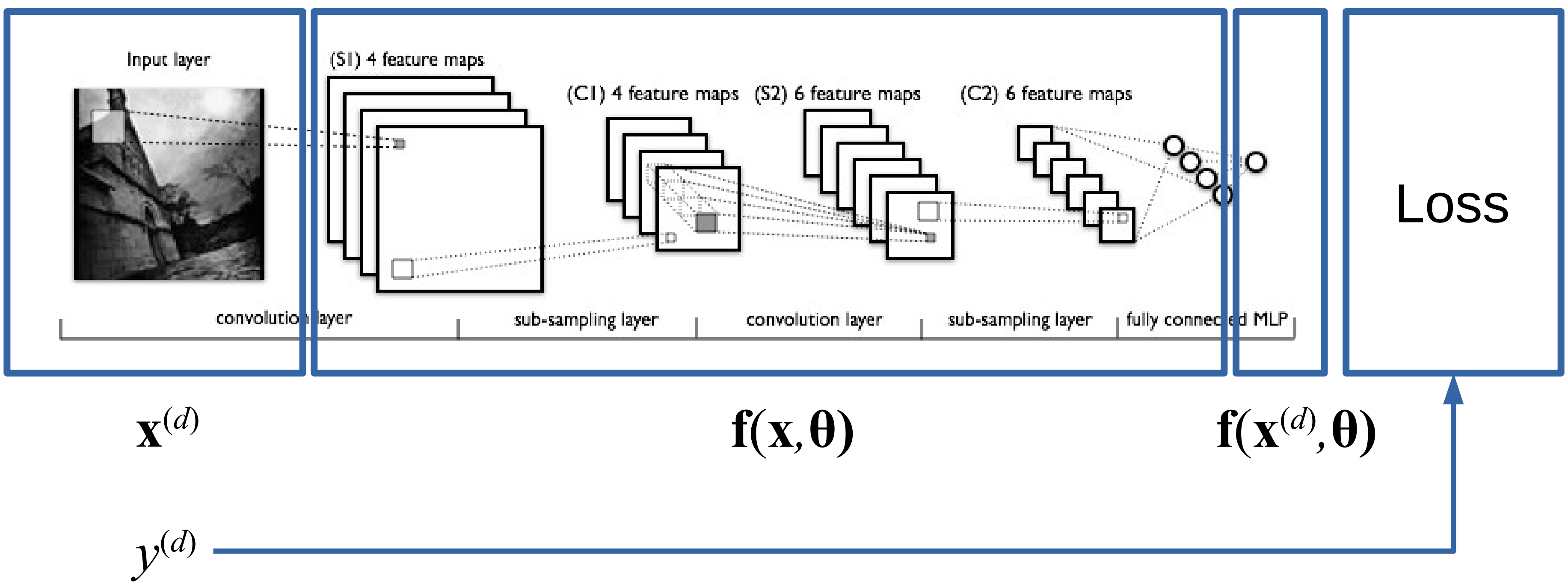


Image from: http://deeplearning.net/

Training neural networks

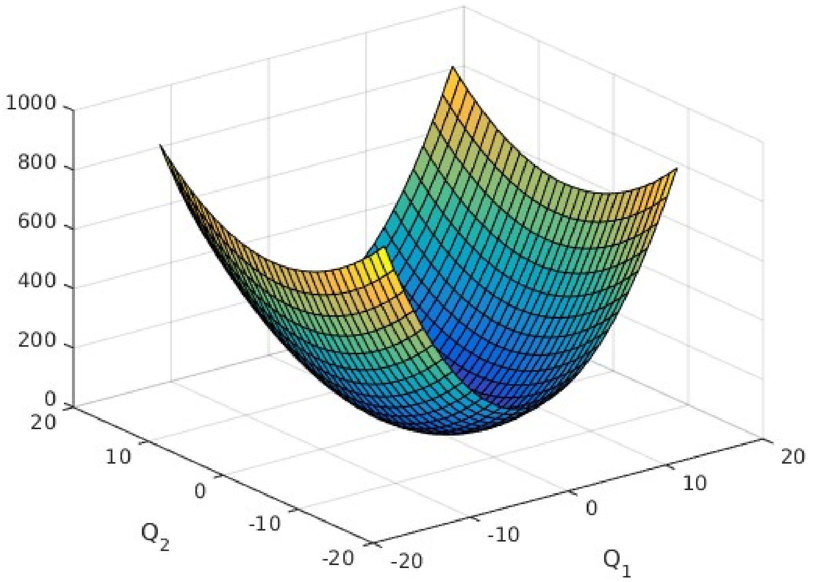
- Minimizing the loss

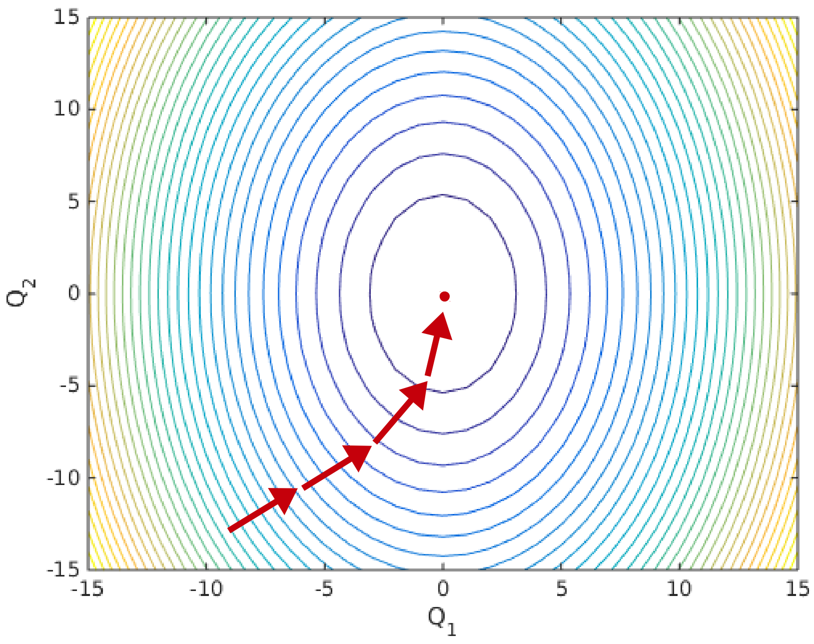
$$\operatorname{argmin}_\theta \frac{1}{D} \sum_t E\left(\mathbf{f}\left(x^{(t)}, \boldsymbol{\theta}\right), y^{(t)}\right) + \lambda \Omega\left(\boldsymbol{\theta}\right)$$

Loss function   Regularization

Gradient descent

- Update the parameters θ in the direction of the steepest descent;





Gradient descent

- Update the parameters $\boldsymbol{\theta}$ in the direction of the steepest descent;

$$\Delta = -\frac{1}{D}\sum_t \nabla_{\boldsymbol{\theta}} E\left(\mathbf{f}\left(x^{(t)};\boldsymbol{\theta}\right), y^{(t)}\right) - \mu \nabla_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \Delta$$

Learning rate

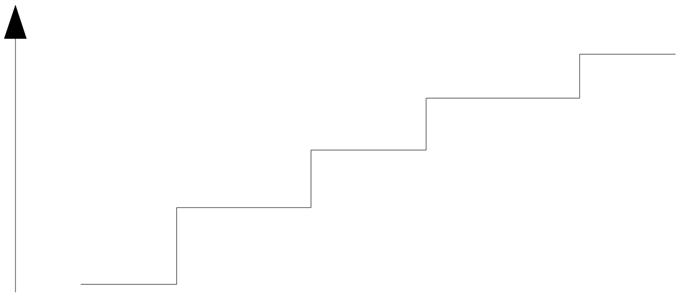
Training neural networks

- What needs to be defined/computed:
  - Loss function
  - Derivative of $\mathrm{E}(\mathbf{f}(\mathbf{x},\boldsymbol{\theta}), y)$ and $\Omega(\boldsymbol{\theta})$ w.r.t every parameter θ_i
    - So we can perform gradient descent
    - This has to be done in an efficient way that scales well for deep networks
  - Initial values for $\boldsymbol{\theta}$

Training neural networks

- What needs to be defined/computed:
  - Loss function
  - Derivative of $\mathrm{E}(\mathbf{f}(\mathbf{x},\boldsymbol{\theta}), y)$ and $\Omega(\boldsymbol{\theta})$ w.r.t every parameter θ_i
    - So we can perform gradient descent
    - This has to be done in an efficient way that scales well for deep networks
  - Initial values for $\boldsymbol{\theta}$

class. error



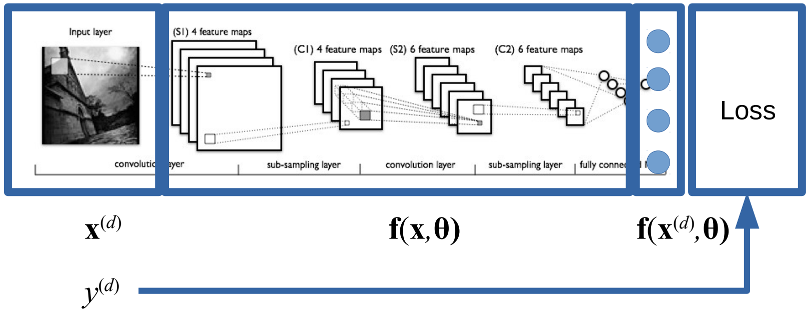
Loss function

- Negative log-likelihood ↔ cross-entropy
  - Minimize the cross-entropy ↔ minimize the uncertainty of the predictions

$$f(\mathbf{x};\boldsymbol{\theta})_c = p(y=c|\mathbf{x};\boldsymbol{\theta})$$

$$E(\mathbf{f}(\mathbf{x}^{(t)};\boldsymbol{\theta}), y^{(t)}) = -\sum_c 1_{y^{(t)}=c} \log p(y=c|\mathbf{x}^{(t)};\boldsymbol{\theta})$$

sum over all classes



Softmax

- Generalization of the logistic function
  - Squashes the inputs to the [0 1] range

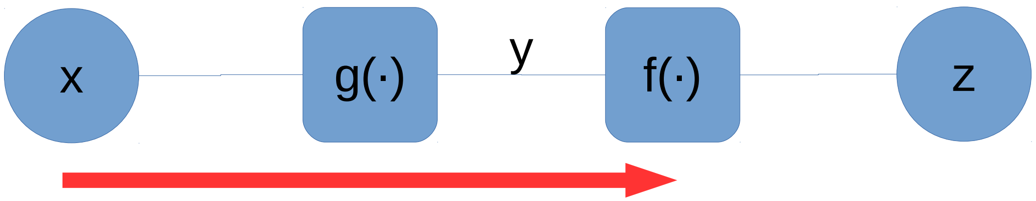
Logistic function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Softmax function: $\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_i e^{z_i}}$

Training neural networks

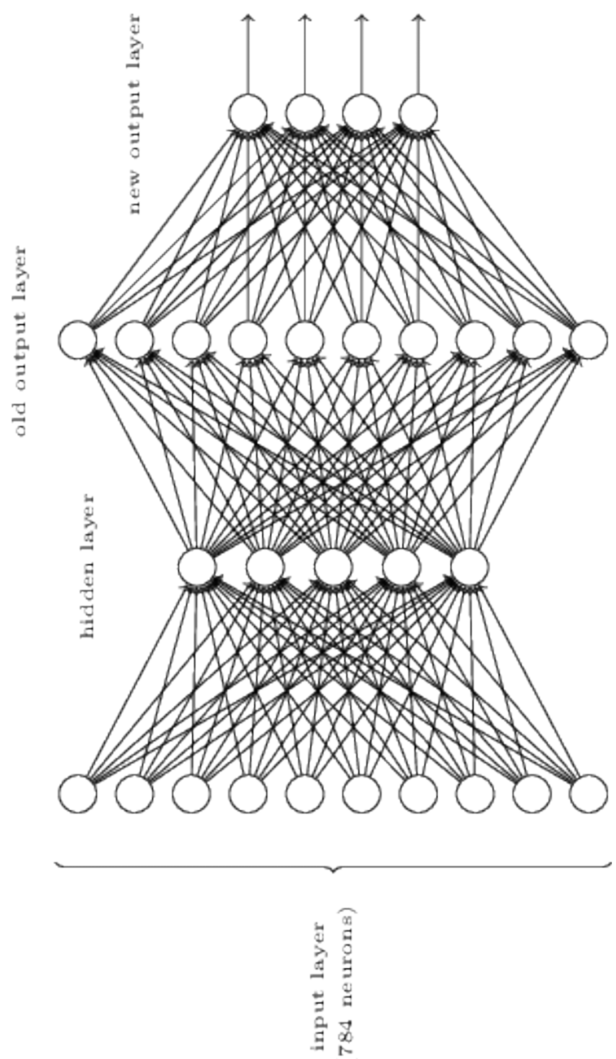
- How to efficiently compute the gradient w.r.t. to every parameter?
- Backpropagation: it's just the chain rule of differentiation applied to neural networks

$$z=f(y); y=g(x); z=f(g(x))$$



$$\frac{dz}{dx}=\frac{dz}{dy}\frac{dy}{dx}$$

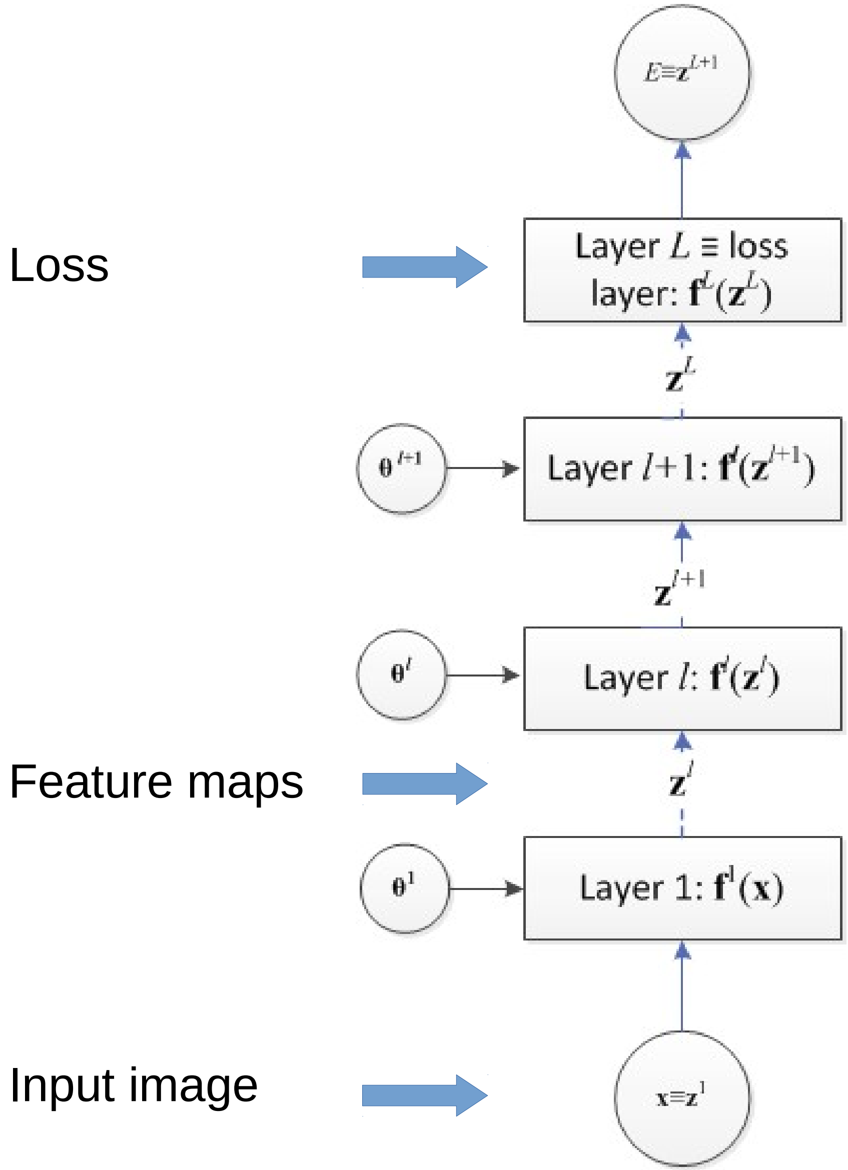
Neural networks: modular approach



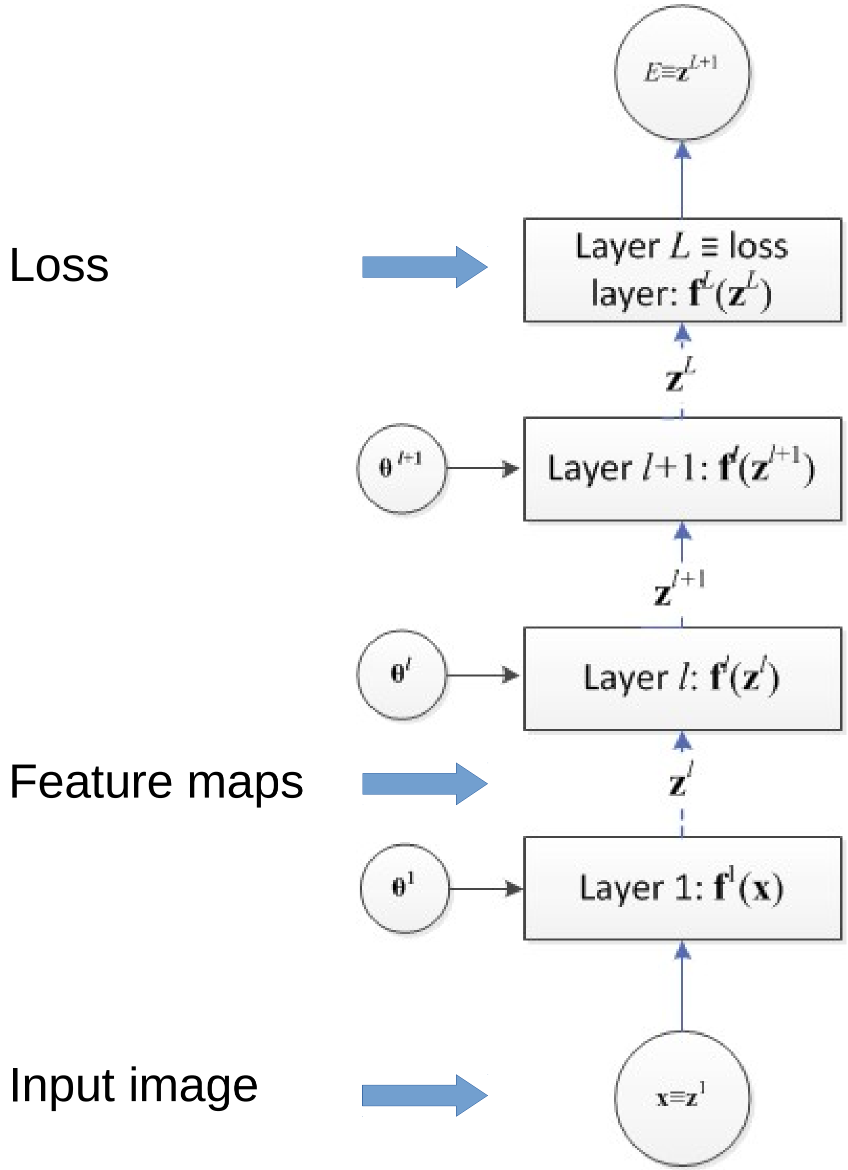
- “Classical” view: individual neurons with nonlinearities
- Better:
- Each layer can be a single entity (module) that computes a function
- The layers have vector inputs, outputs and (sometimes) parameters
- This is how NNs are implemented in code

Image from: http://lauraskelton.github.io/

Neural networks: modular approach



Neural networks: modular approach



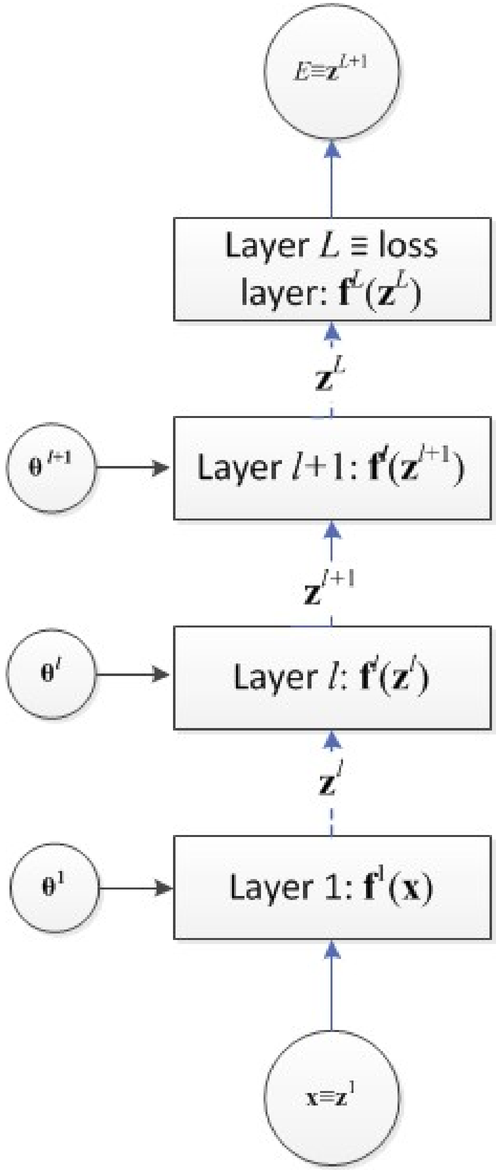
$$E = \mathbf{f}^L(\mathbf{z}^L)$$

$$\mathbf{z}^L = \mathbf{f}^{L-1}(\mathbf{z}^{L-1})$$

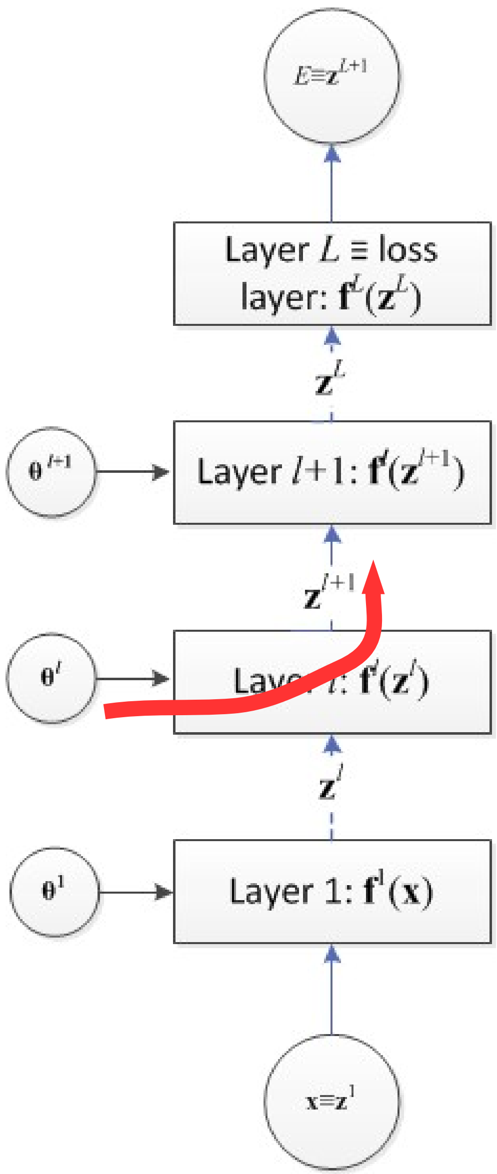
...

$$E = \mathbf{f}^L(\mathbf{f}^{L-1}(\ldots(\mathbf{f}^l(\mathbf{z}^l))))$$

Neural networks: modular approach

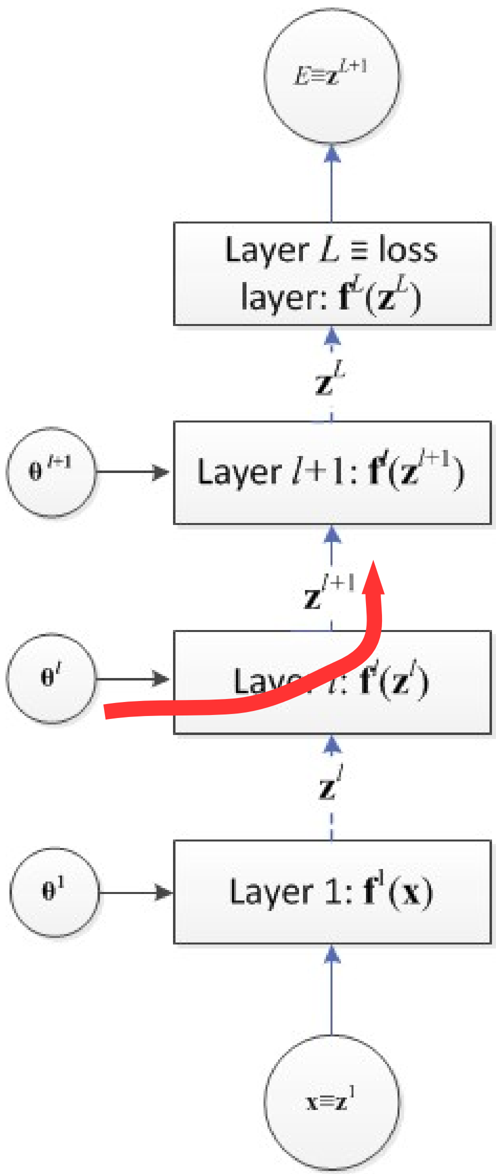


Neural networks: modular approach



$$\frac{\partial E}{\partial \boldsymbol{\theta}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \boldsymbol{\theta}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{f}^l(\mathbf{z}^l, \boldsymbol{\theta}^l)}{\partial \boldsymbol{\theta}^l}$$

Neural networks: modular approach

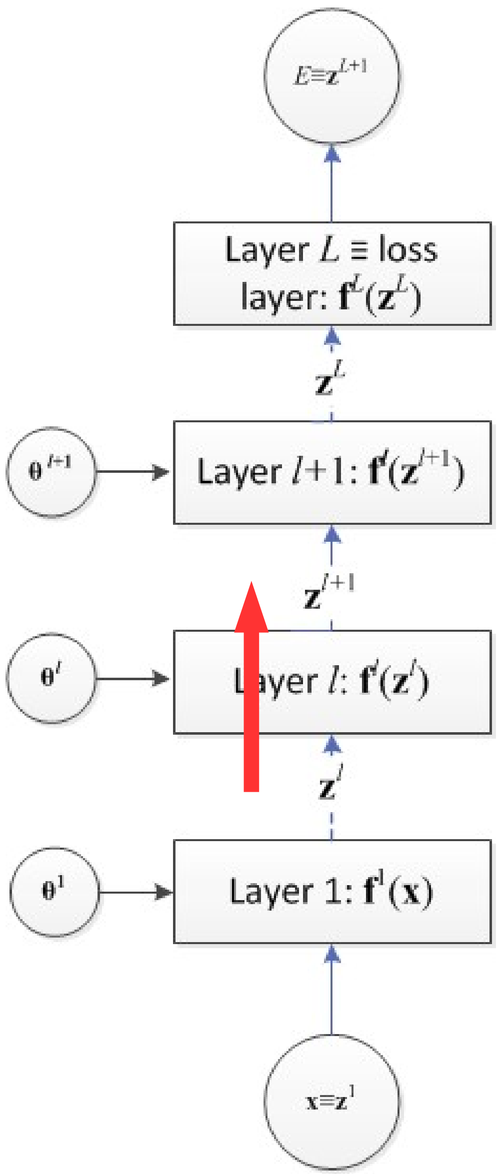


$$\frac{\partial E}{\partial \boldsymbol{\theta}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \boldsymbol{\theta}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{f}^l(\mathbf{z}^l, \boldsymbol{\theta}^l)}{\partial \boldsymbol{\theta}^l}$$

$$\frac{\partial \mathbf{f}^l(\mathbf{z}^l, \boldsymbol{\theta}^l)}{\partial \boldsymbol{\theta}^l}$$ ✓

$$\frac{\partial E}{\partial \mathbf{z}^{l+1}}$$?

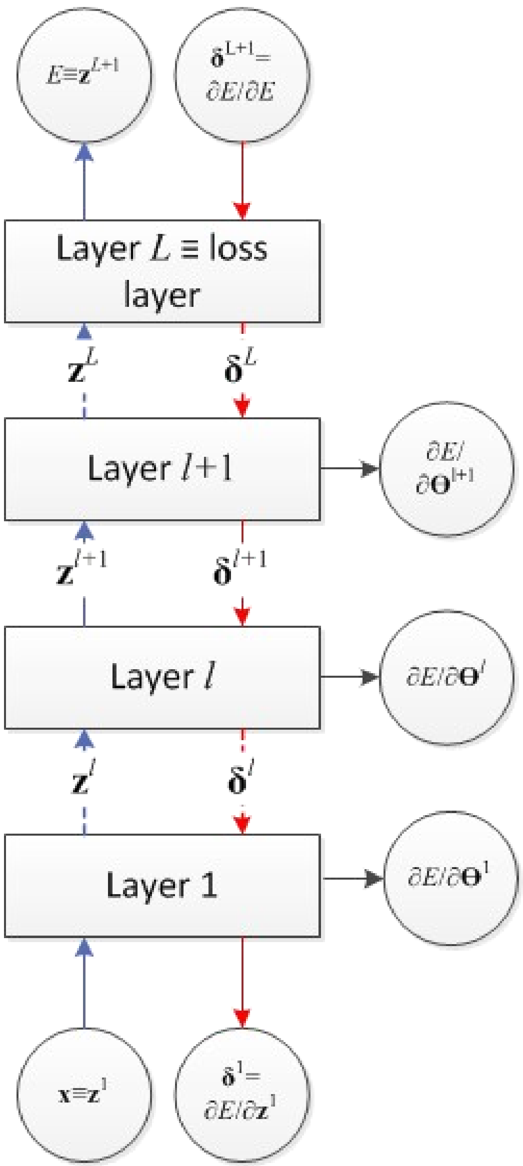
Neural networks: modular approach



$$\frac{\partial E}{\partial \mathbf{z}^l} = \boldsymbol{\delta}^l = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l}$$

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l, \boldsymbol{\theta}^l)}{\partial \mathbf{z}^l} \quad \checkmark$$

Neural networks: modular approach



- Forward computation:

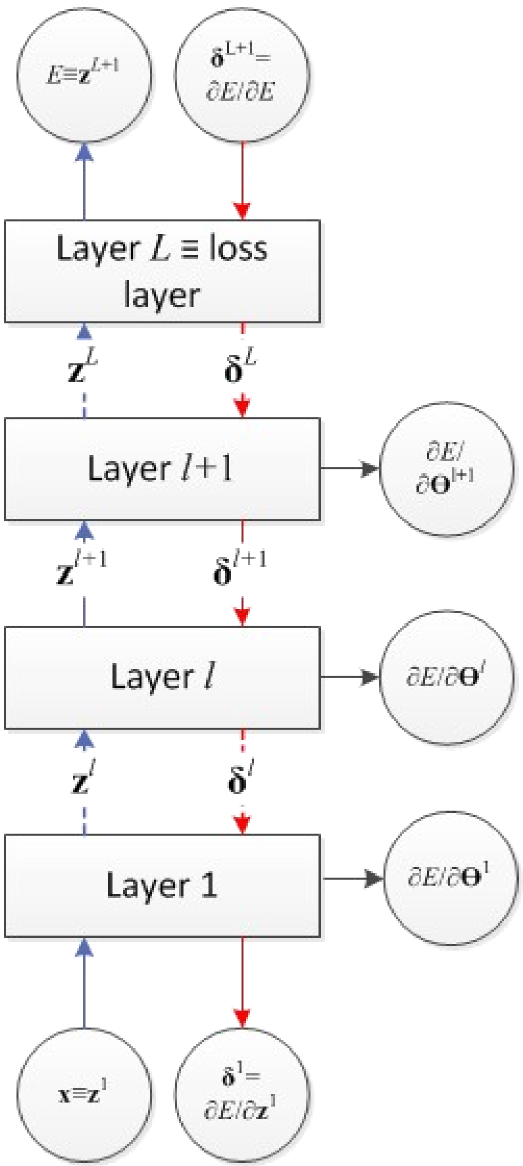
$$\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta})$$

- Backward computation:

$$\frac{\partial E}{\partial \boldsymbol{\theta}^l} = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial \boldsymbol{\theta}^l}$$

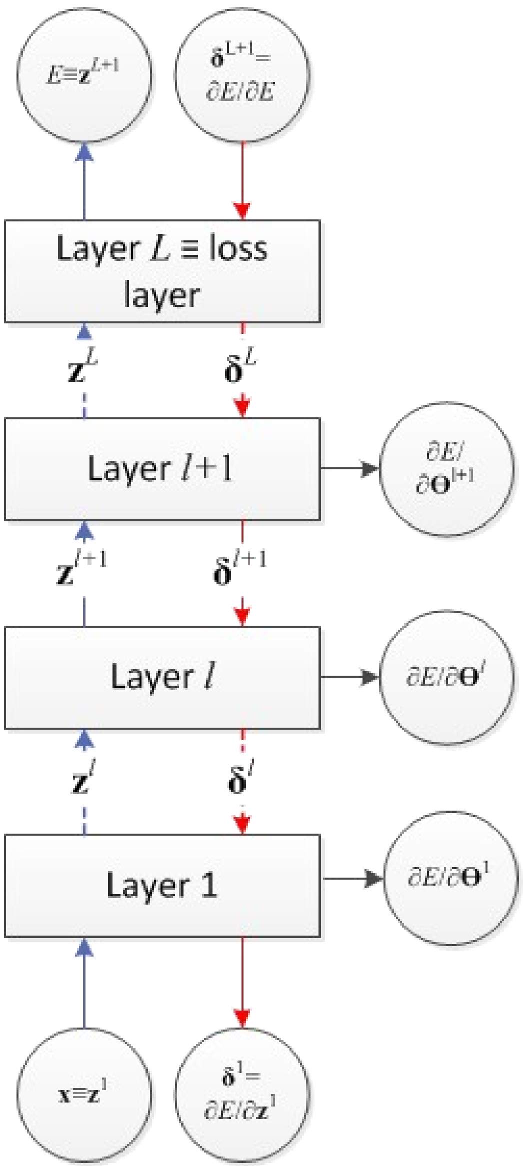
$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial \mathbf{z}^l}$$

Neural networks: modular approach



- Forward computation:
  - Given the input, compute the output of every layer
- Backward computation:
  - Given the input and the gradient term term passed from the layer above:
    - Compute the derivatives of the loss w.r.t. to the parameters of the layer
    - Compute the gradient term for the current layer

Neural networks: modular approach

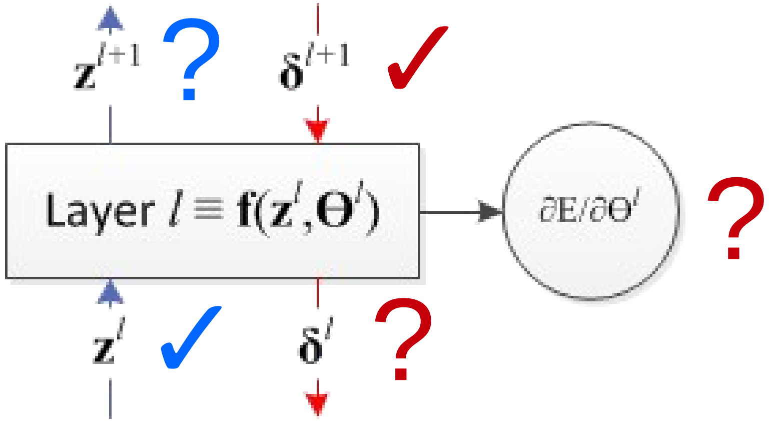


- Forward computation:
  - Given the input, compute the output of every layer
- Backward computation:
  - Given the input and the gradient term term passed from the layer above:
    - Compute the derivatives of the loss w.r.t. to the parameters of the layer
    - Compute the gradient term for the current layer

What we really need

“Helper”

Implementing a module



- Forward computation:

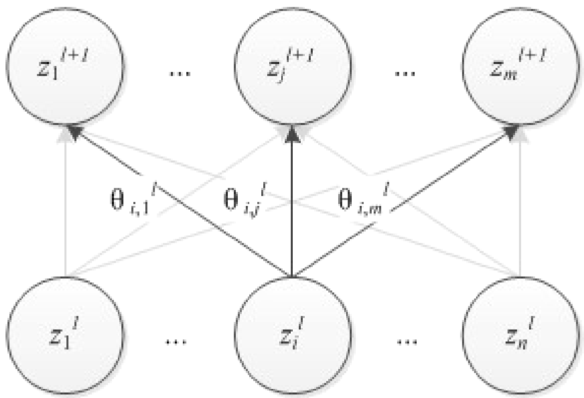
$$\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta})$$

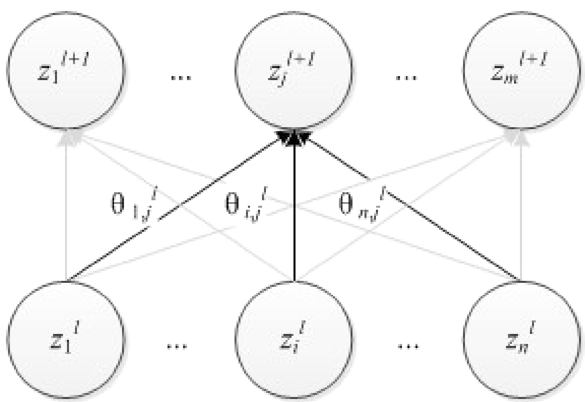
- Backward computation:

$$\frac{\partial E}{\partial \boldsymbol{\theta}^l} = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \theta^l)}{\partial \boldsymbol{\theta}^l}$$

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \theta^l)}{\partial \mathbf{z}^l}$$

Example: linear layer





- Forward computation:

$$z_j^{l+1} = \sum_i z_i \theta_{i,j}^l$$

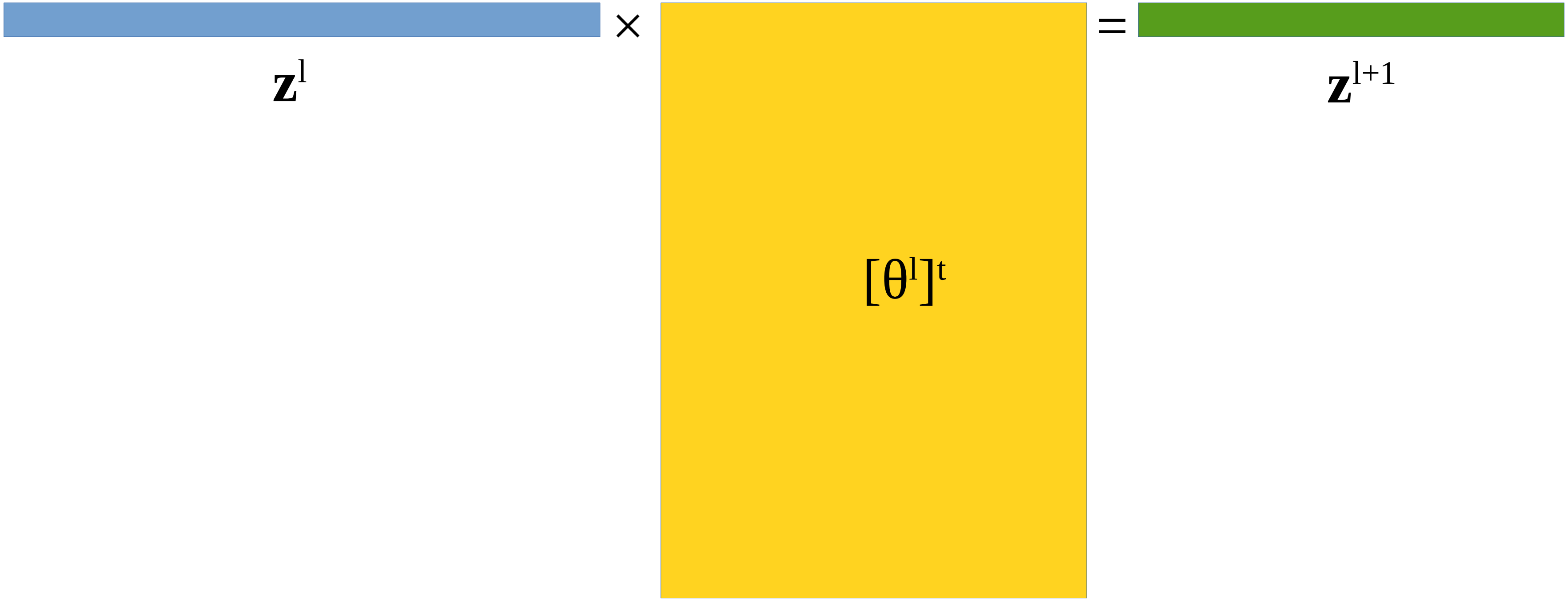
- Backward computation:

$$\frac{\partial E}{\theta_{i,j}^l} = \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\theta_{i,j}^l} = \delta_j^{l+1} z_i^l$$

$$\delta_i^l = \frac{\partial E}{\partial z_i^l} = \sum_j \frac{\partial E}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \theta_{i,j}^l$$

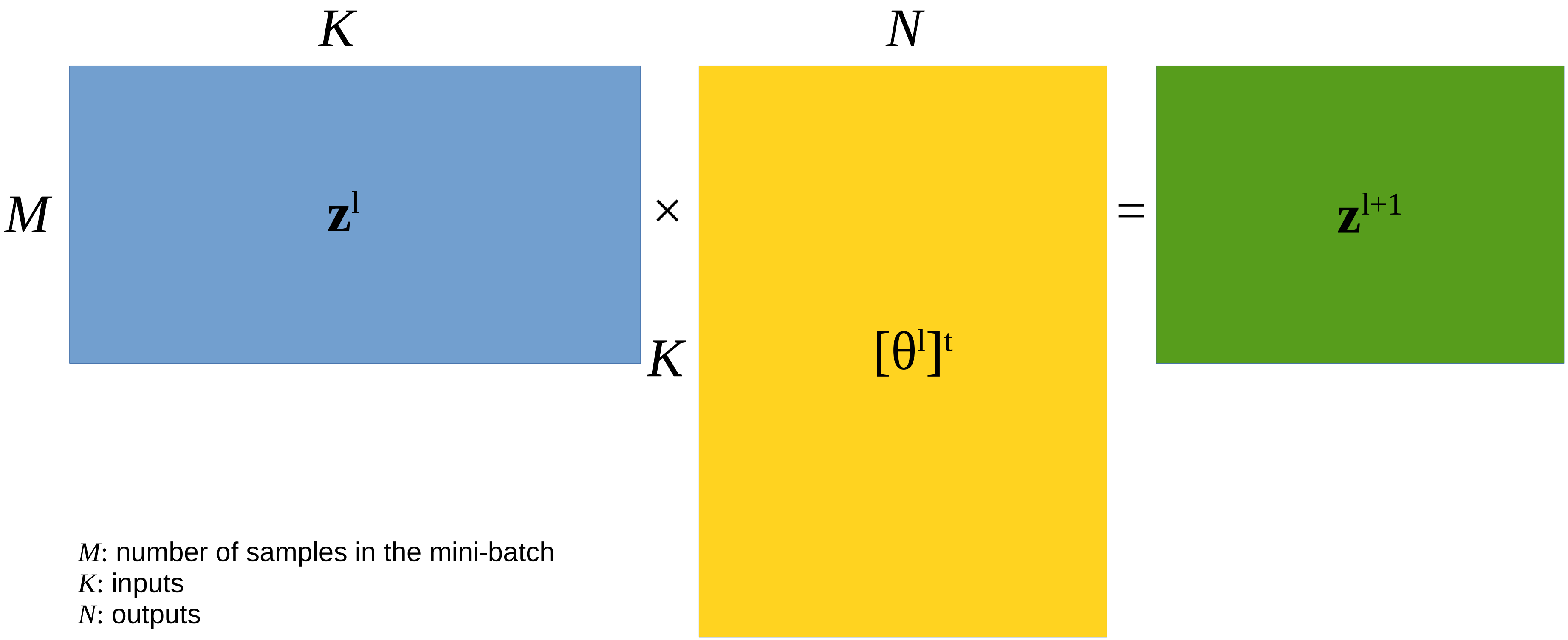
Example: linear layer

- Forward computation, single sample
  - Ignoring biases



Example: linear layer

- Forward computation, more than one sample



M: number of samples in the mini-batch
K: inputs
N: outputs

Example implementation: linear layer in caffe

- Forward computation:

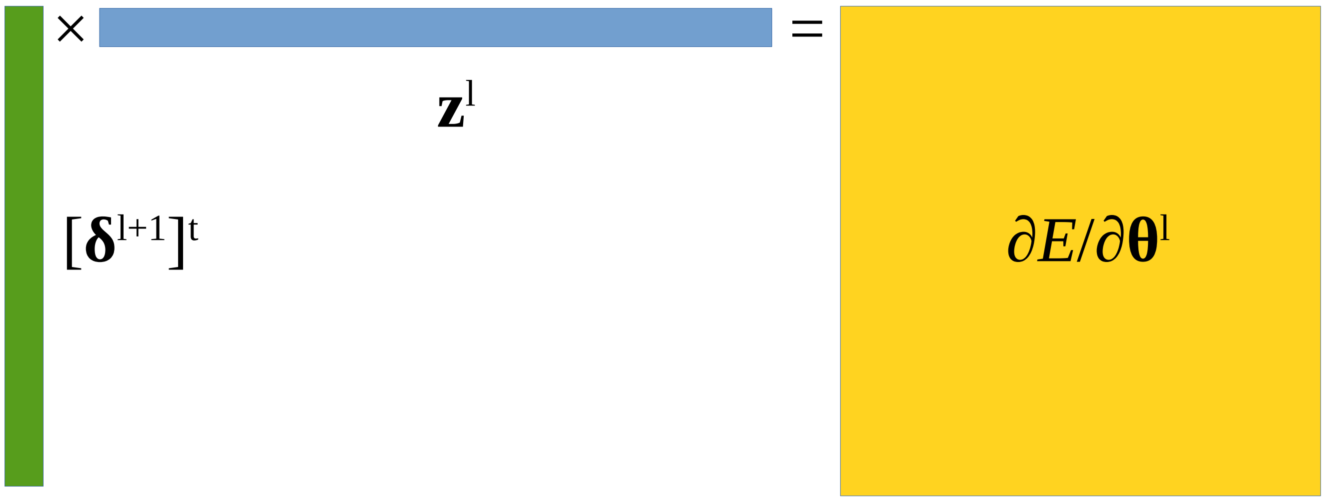
```
template <typename Dtype>
void InnerProductLayer<Dtype>::Forward_cpu(
    const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top)
{
  const Dtype* bottom_data = bottom[0]->cpu_data();
        Dtype* top_data    = top[0]->mutable_cpu_data();
  const Dtype* weight      = this->blobs_[0]->cpu_data();

  caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasTrans, M_, N_, K_, (Dtype)1.,
      bottom_data, weight, (Dtype)0., top_data);
...
}
```

`gemm` ≡ general matrix multiplication (BLAS function):

$C \leftarrow \alpha\, \mathbf{AB} + \beta\, \mathbf{C}$

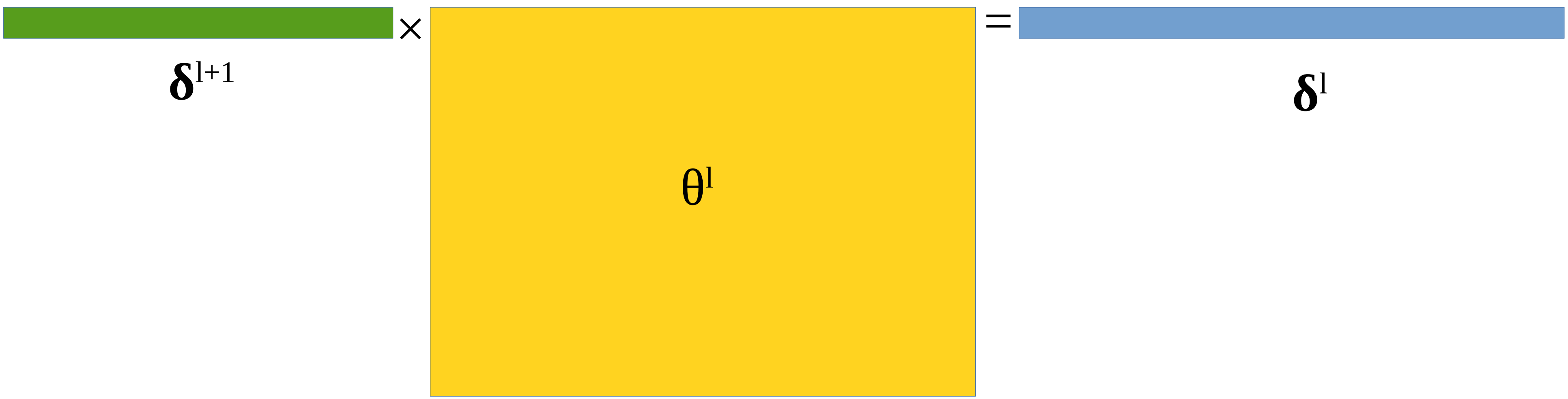
Example: linear layer

- Backward computation
  - Gradient w.r.t. parameters



Example: linear layer

- Backward computation
  - Gradient w.r.t. input



Example implementation: linear layer in caffe

- Backward computation:

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Backward_cpu(
    const vector<Blob<Dtype>*>& top, const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom)
{
  if (this->param_propagate_down_[0]) {
    const Dtype* top_diff = top[0]->cpu_diff();
    const Dtype* bottom_data = bottom[0]->cpu_data();
    // Gradient with respect to weight
    caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans, N_, K_, M_, (Dtype)1.,
        top_diff, bottom_data, (Dtype)1., this->blobs_[0]->mutable_cpu_diff());
  }
...
  if (propagate_down[0]) {
    const Dtype* top_diff = top[0]->cpu_diff();
    // Gradient with respect to bottom data
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, K_, N_, (Dtype)1.,
        top_diff, this->blobs_[0]->cpu_data(), (Dtype)0., bottom[0]->mutable_cpu_diff());
  }
}
```

Regularization

- L2 regularization
  - Penalize the square of the weights
  - Keeps the weights small
  - A.k.a. weight decay

$$\operatorname{argmin}_{\theta} \frac{1}{D} \sum_{d} E\left(\mathbf{f}\left(x^{(d)}, \boldsymbol{\theta}\right), y^{(d)}\right) + \lambda \Omega(\boldsymbol{\theta})$$

$$\Omega(\theta) = \frac{1}{2} \sum_{l} \sum_{i} \sum_{j} \left(\theta_{i,j}^{l}\right)^{2}$$

$$\frac{\partial \Omega(\theta)}{\partial \theta_{i,j}^{l}} = \theta_{i,j}^{l}$$

Regularization

- L1 regularization
  - Penalize the absolute values of the weights
  - Keeps the weights small, leads to sparse weights

$$\Omega(\theta) = \frac{1}{2} \sum_{l} \sum_{i} \sum_{j} \left| \left(\theta_{i,j}^{l} \right) \right|$$

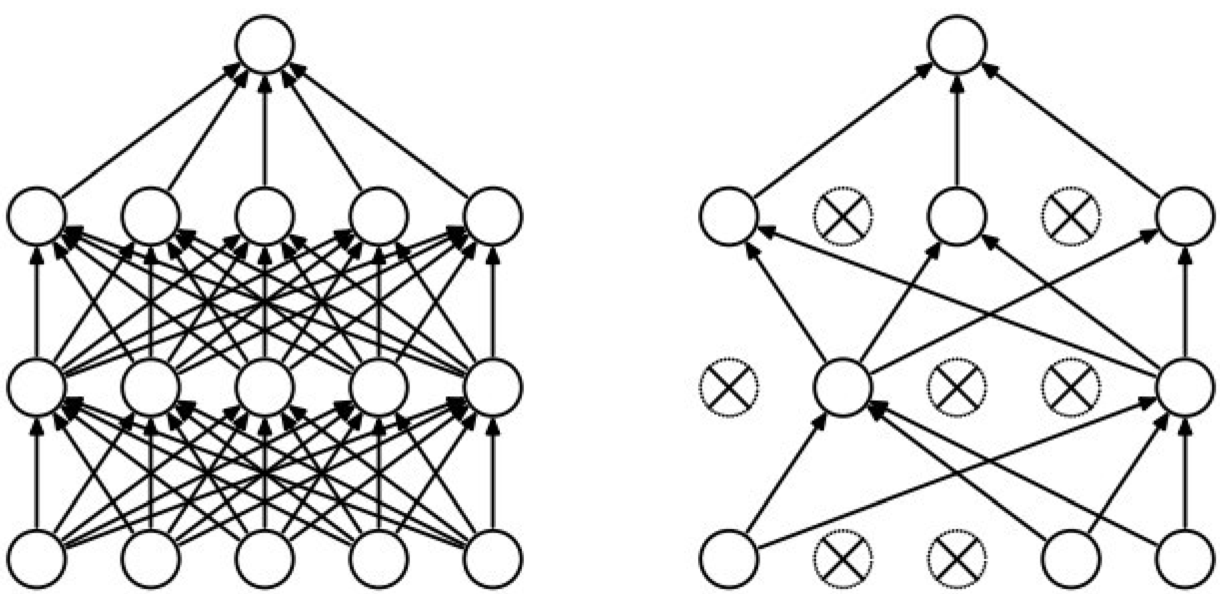
$$\frac{\partial \Omega(\theta)}{\partial \theta_{i,j}^{l}} = \operatorname{sign}\left(\theta_{i,j}^{l} \right)$$

Regularization

- Other approaches to combat over-fitting
  - Smaller architectures
  - Data augmentation
  - Dropout
  - Early stopping

Dropout

- During training, randomly “turn off” some neurons
- Prevents co-adaptation

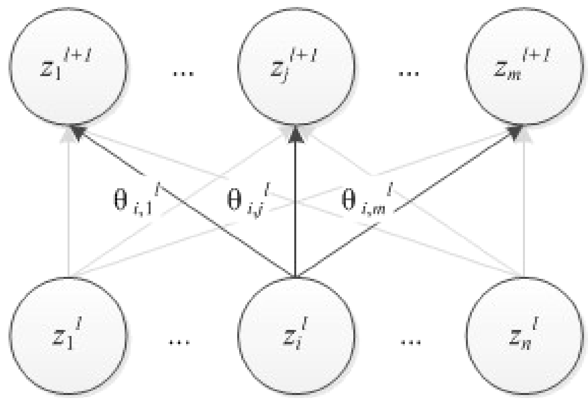


(a) Standard Neural Net (b) After applying dropout.

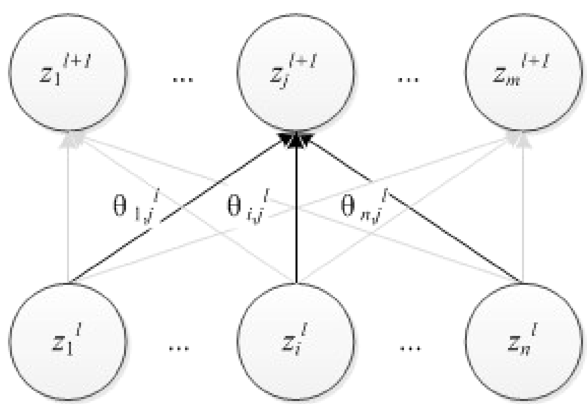
Image from: http://cs231n.github.io/

Parameter initialization

fan-out



fan-in



- Weights
  - Small random numbers usually drawn from Gaussian or uniform distribution
  - Heuristic for the scale of the weights that prevents the “signal” from shrinking as it propagates trough the network:

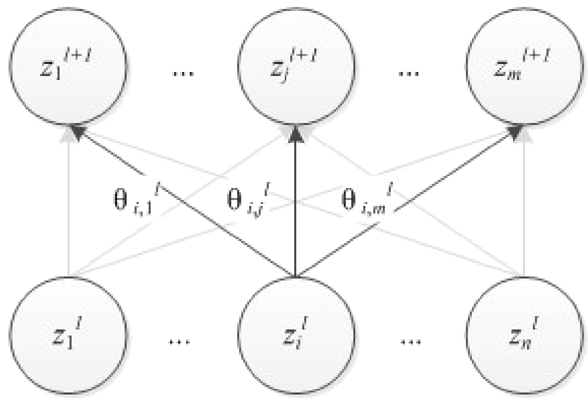
$$\mathrm{var}\left(\boldsymbol{\theta}^{\mathbf{l}}\right)=\frac{1}{n_{\text{fan in}}}$$

$$\mathrm{var}\left(\boldsymbol{\theta}^{\mathbf{l}}\right)=\frac{2}{n_{\text{fan in}}+n_{\text{fan out}}}$$

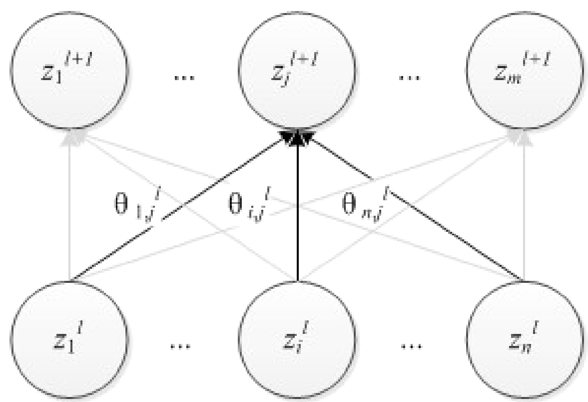
Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International conference on artificial intelligence and statistics. 2010.

Parameter initialization

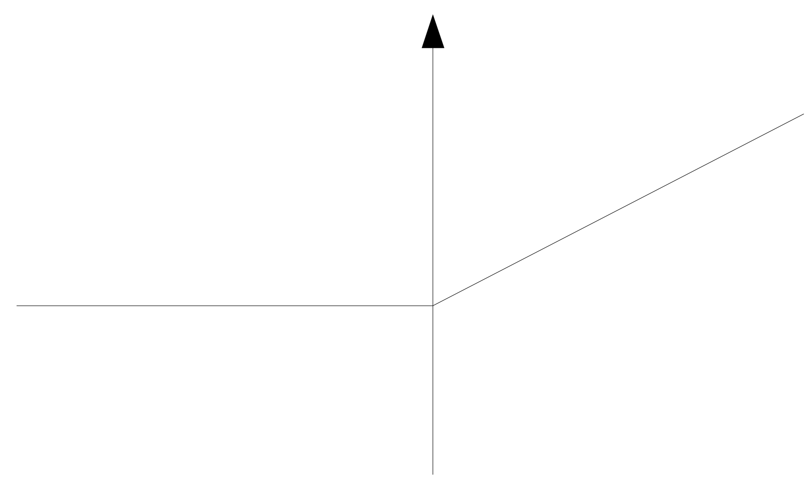
fan-out



fan-in



- Biases
  - Usually initialized to 0
  - When using ReLU nonlinearities consider values >0 to avoid “dead” units



Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International conference on artificial intelligence and statistics. 2010.

Stochastic gradient descent

Batch: $$\Delta = -\frac{1}{D}\sum_t \nabla_{\boldsymbol{\theta}} E\left(\mathbf{f}\left(x^{(t)};\boldsymbol{\theta}\right), y^{(t)}\right) - \mu \nabla_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$

$$\theta \leftarrow \theta + \eta \Delta$$

Stochastic: $$\Delta_s = \nabla_{\boldsymbol{\theta}} E\left(\mathbf{f}\left(x^{(t)};\boldsymbol{\theta}\right), y^{(t)}\right) - \mu \nabla_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$

Mini-batch: $$\Delta_s = -\frac{1}{M}\sum_m \nabla_{\boldsymbol{\theta}} E\left(\mathbf{f}\left(x^{(m)};\boldsymbol{\theta}\right), y^{(m)}\right) - \mu \nabla_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$

$$M \ll D$$

Momentum

$$\Delta_s = -\frac{1}{M} \sum_m \nabla_{\boldsymbol{\theta}} E\left(\mathbf{f}\left(x^{(m)}; \boldsymbol{\theta}\right), y^{(m)}\right) - \mu \nabla_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$

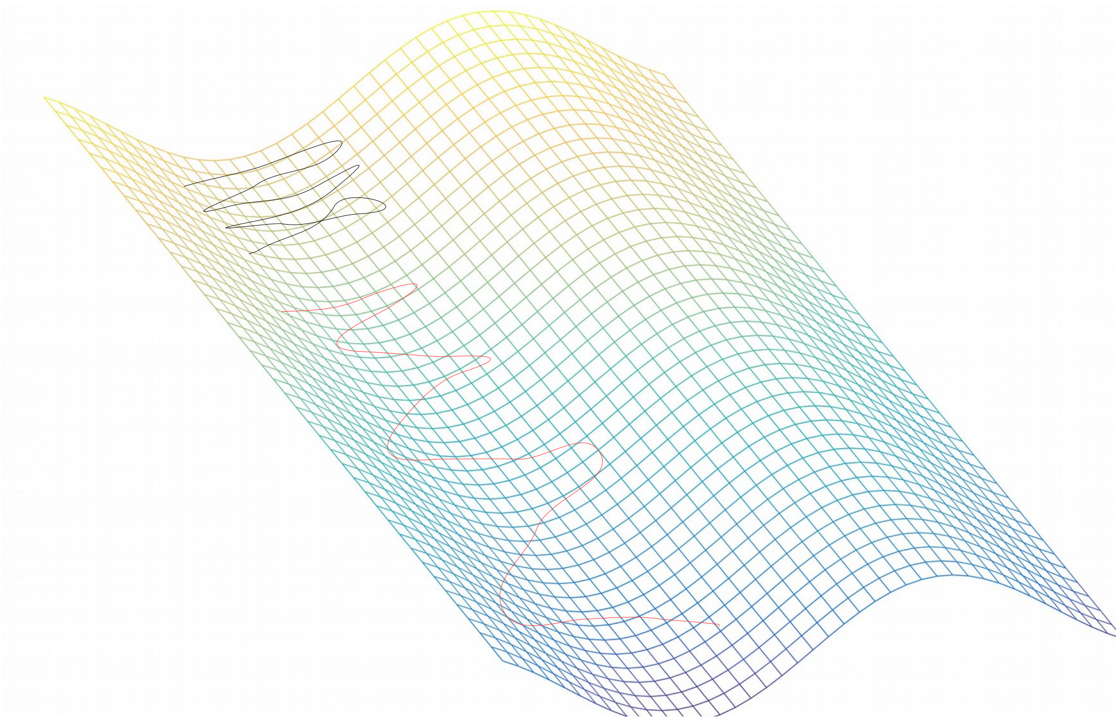
$$\mathbf{v} = \mu \mathbf{v} + \eta \Delta_s$$

Momentum

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$$

Instead of:

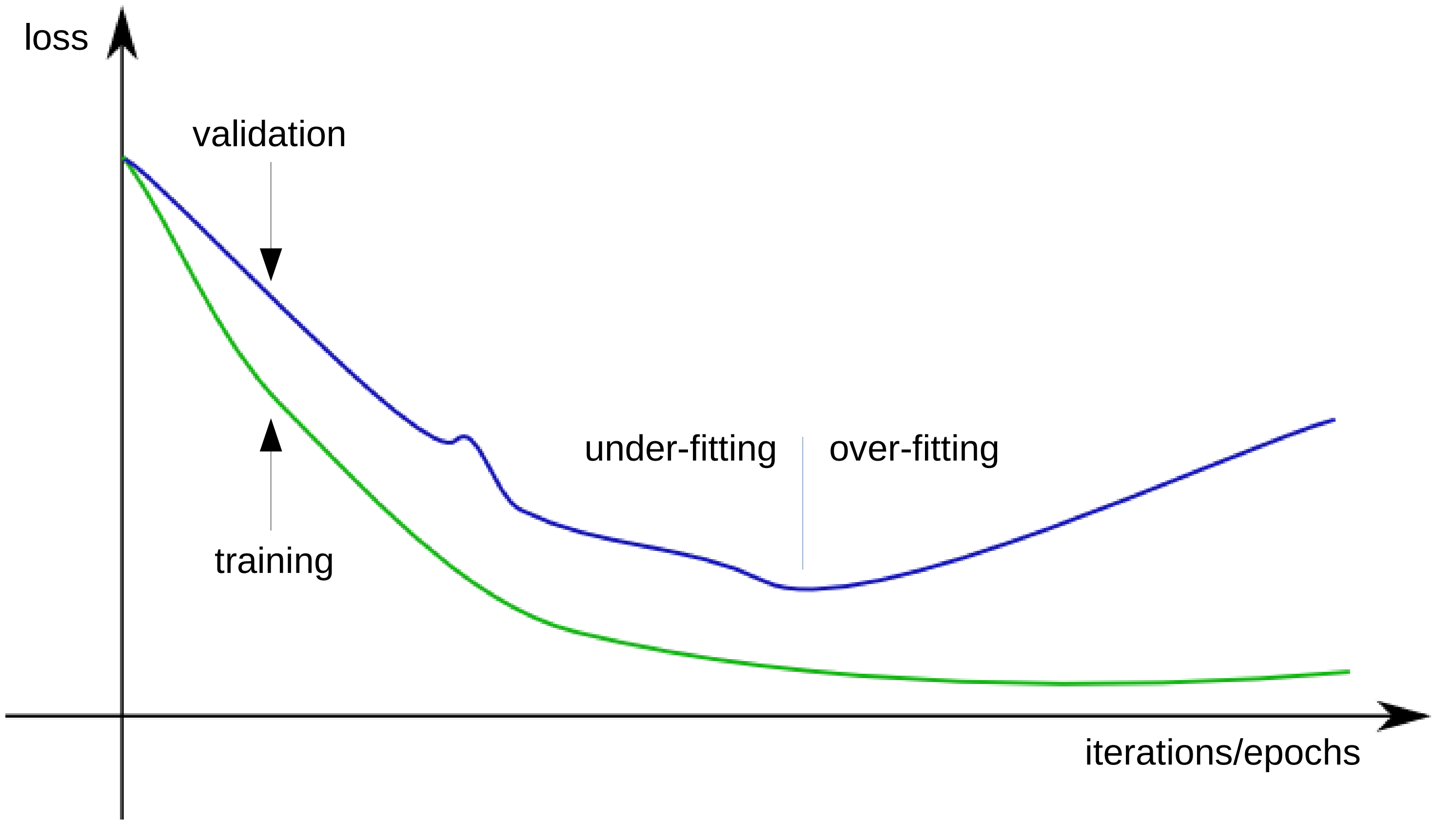
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \Delta_s$$



Observing the training process

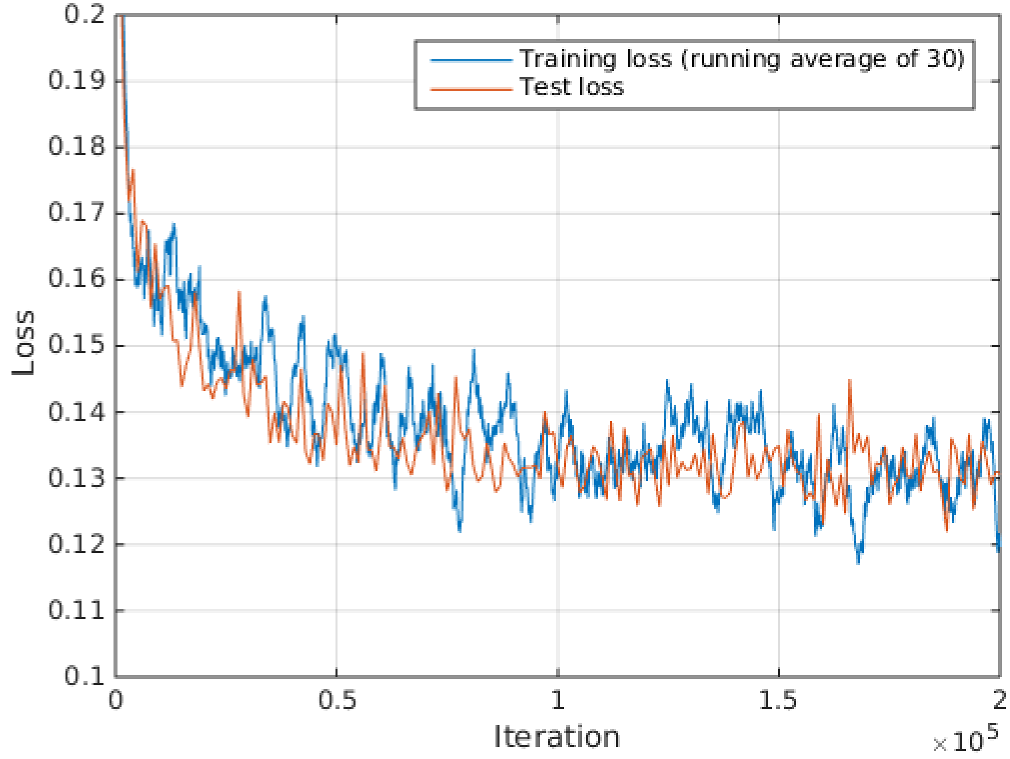
- Data split:
  - Training subset: used during the optimization
  - Validation subset: used to monitor overfitting, select meta-parameters
  - Testing dataset: used to evaluate the performance of the trained model

Observing the training process



Observing the training process

- A more realistic example (with excellent generalization)



Choosing the learning rate

- Too small: slow to reach a minima
- Too large: may oscillate around a minima

Random performance, for softmax: $\ln(num\_classes)$

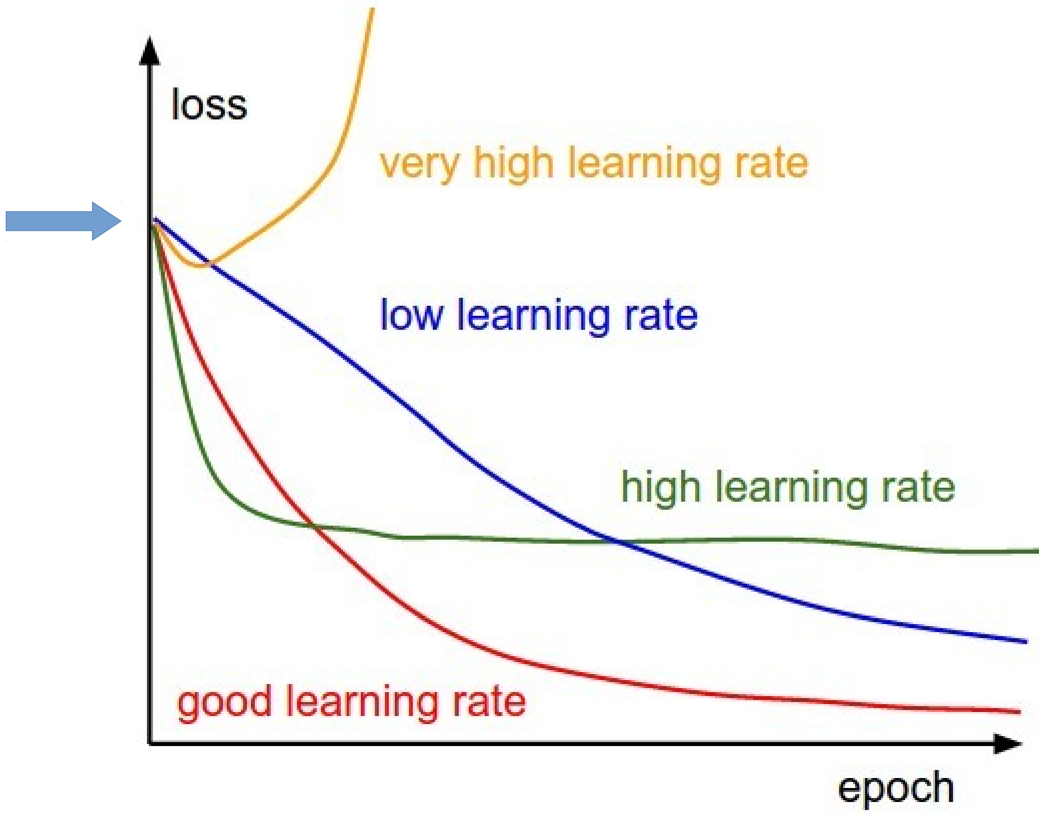
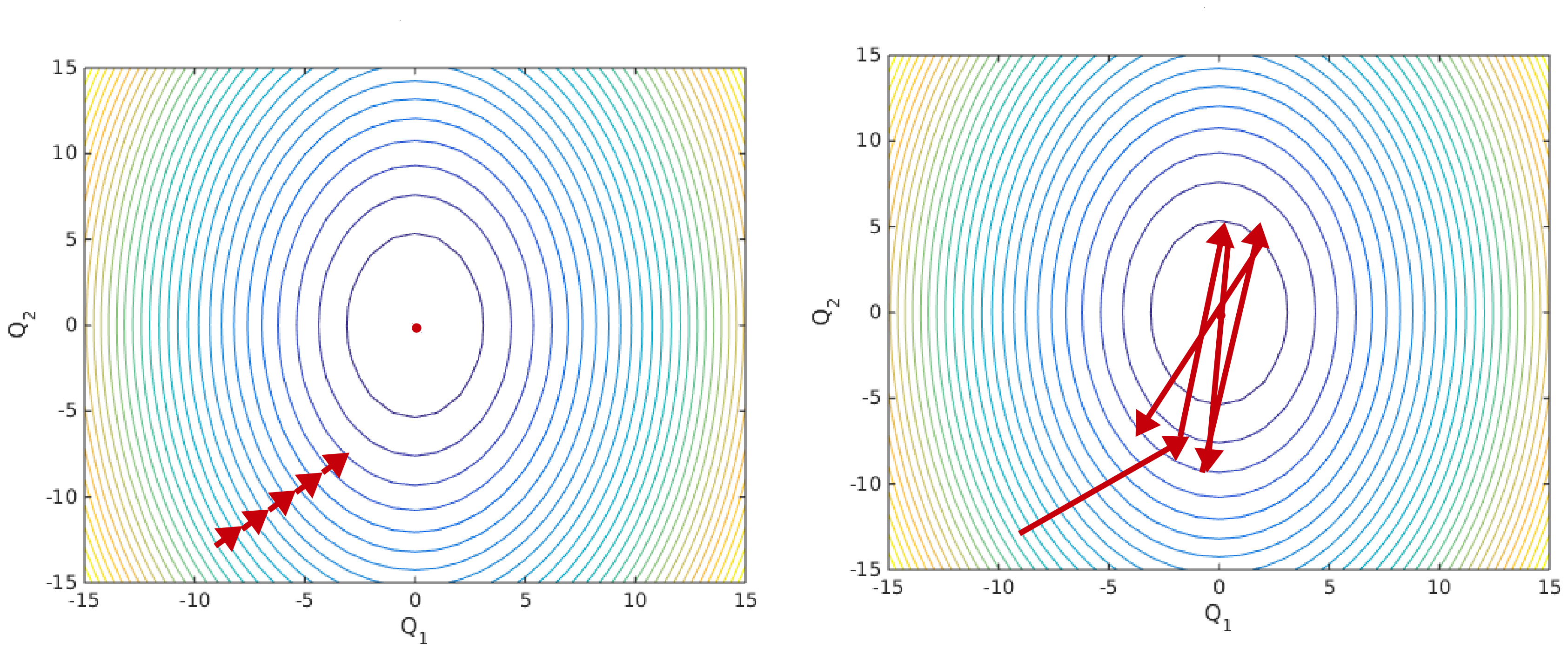


Image from: http://cs231n.github.io/

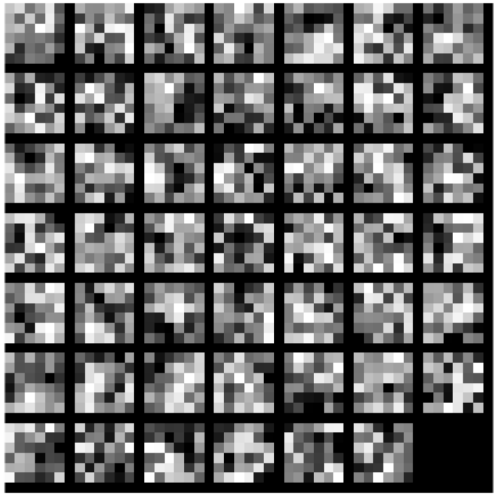
Choosing the learning rate

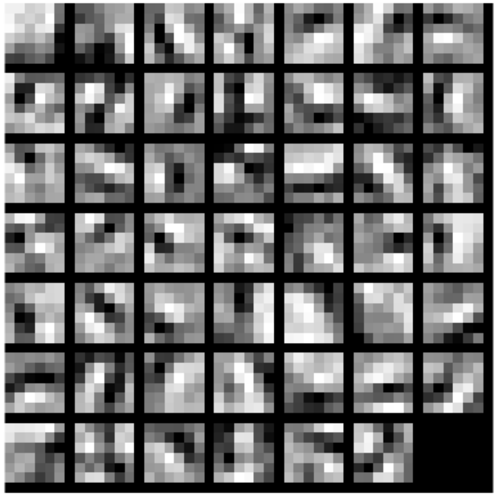
- Too small: slow to reach a minimum
- Too large: may oscillate around a minimum



Observing the learned features

- The lower-level features should be “smooth” and have structure





- Data augmentation/normalization
  - Explore invariances of your data/problem
  - Subtract mean, whitening (decorelate the input variables)
- Network architecture
- Loss function
  - NLL with softmax for most classification problems
- Weight initialization
  - Based on the number of inputs/outputs to the layer
- Regularization strategy
  - Weight decay (L2 is the most common choice), dropout, early stopping
- Gradient descent method
  - SGD with momentum in most cases
- Learning rate
- Observe the training process and make necessary adjustments
  - Consider an optimization strategy (such as grid or random search) for the meta-parameters (learning rate, weight decay etc.)

Recommended resources

- Oxford Machine Learning Course from Nando de Freitas
  - https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/
- Stanford Convolutional Neural Networks for Visual Recognition Course from Andrej Karpathy
  - http://cs231n.github.io/
- Nural Networks Online Course from Hugo Larochelle
  - http://info.usherbrooke.ca/hlarochelle/neural_networks/content.html
- Caffe documentation and examples
  - http://caffe.berkeleyvision.org/