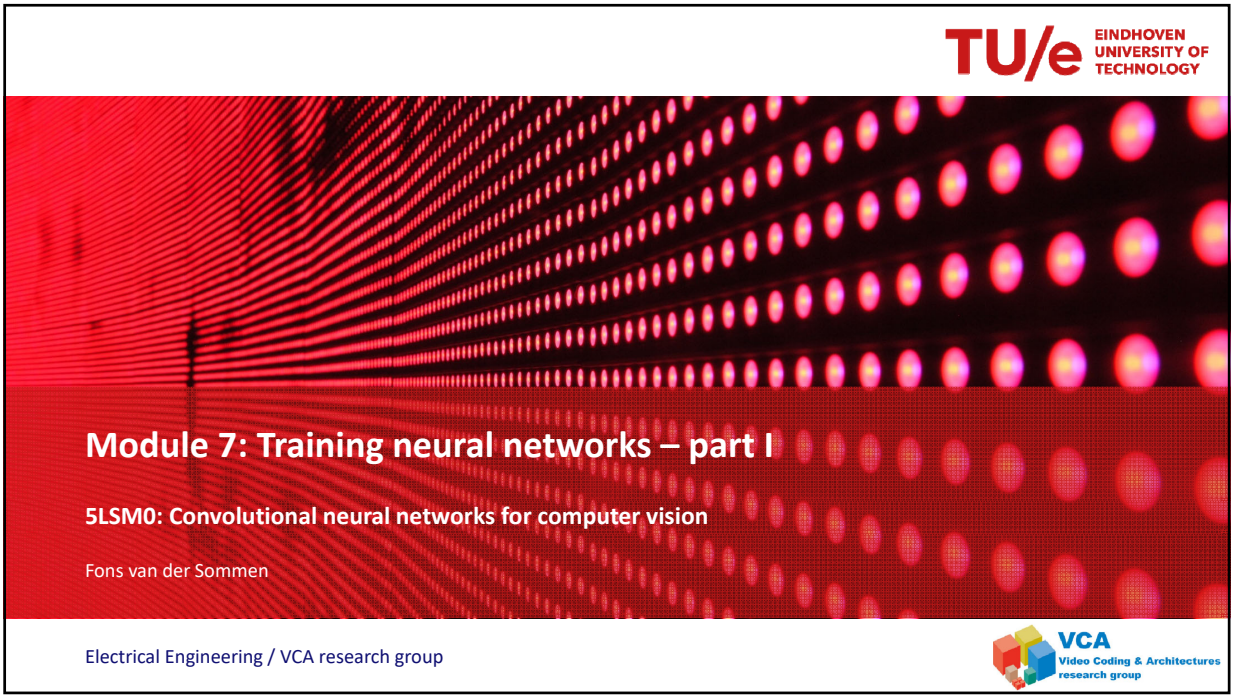


Module 7: Training neural networks – part I

5LSM0: Convolutional neural networks for computer vision

Fons van der Sommen

Electrical Engineering / VCA research group

Administrative

Practical assigments (20%)

- Assigment 1 (10%) due Monday March 11th 23:59 CET
- Assigment 2 (10%) follows in that same week (week #11)

Current plan:

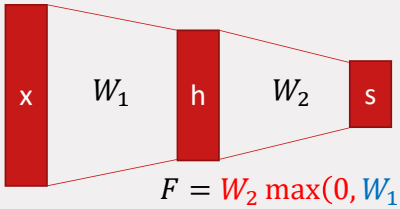
- Written exam (35%)
- Final assignment (35%)

Paper reading assignment (10%)

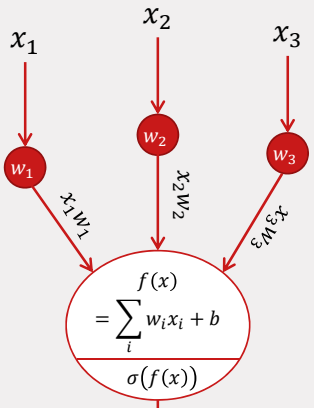
- Choose paper from pre-defined list or propose one yourself
- Carefully read the paper in groups of 2 (/ 3)
- Present the paper during one of the slots of the lecture hours (10 mins + 2mins Q&A)
  - *If you're unable to make the lecture hours, please contact me (fvdsommen@tue.nl)*
- At least describe the methods, the novelty of the proposed solution and the results
  - *Take turns in presenting*

Last time(s)

Neural Networks

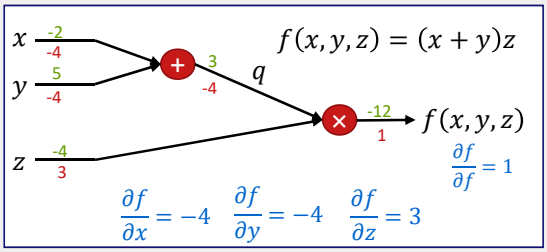


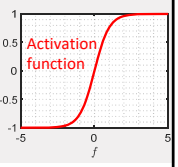
$F = W_2 \max(0, W_1 x)$



Artificial neurons

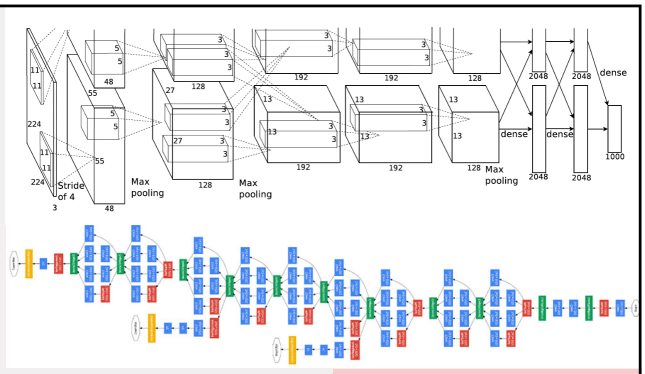
Computational graphs

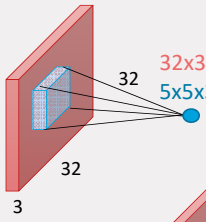


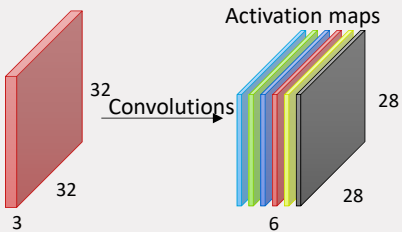


Last time(s)

Convolutional Neural Networks (CNNs)







Different architectures

1	1	2	4
5	6	8	7
3	2	1	0
1	2	3	4

Max pooling

6	8
3	4

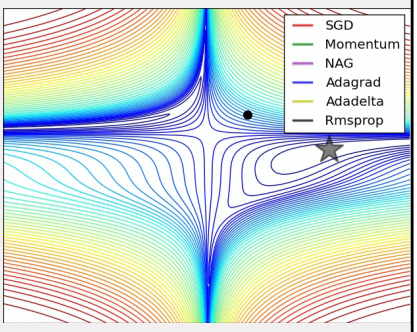
Last time(s)

Stochastic (mini-batch) gradient descent

Classification loss

$$L_i = -\log P(Y = k|X = x_i) = -\log\left(\frac{e^{s_k}}{\sum_j e^{s_j}}\right)$$

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



Learning parameters using optimization

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

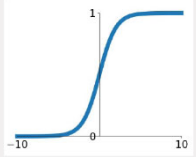
Last time(s)

Chain rule

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

Activation functions

Sigmoid

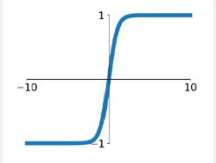


$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Not zero-centered
= inefficient gradient updates

Saturation region
= kills gradient

tanh



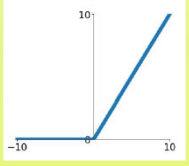
$\tanh x$

(tanh and exp = expensive)

Saturates for $x < 0$
= kills gradient

Go-to activation function

ReLU



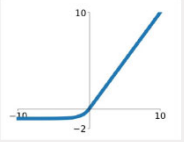
$\max(0, x)$

Cheap and converges fast!

Maxout

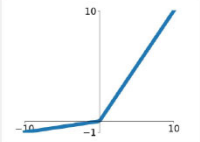
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU



$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Leaky ReLU



$\max(\alpha x, x)$

Fixed $\alpha \in \mathbb{R}^+$

This time

It's all about training!

- Can I feed raw data into my network?
- Where should I start in my loss landscape? (Weight initialization)
- How can I see if the training goes well?
- How should I pick my hyperparameters?
- Some other fancy tricks to improve training (e.g. Batch normalization)

Data normalization

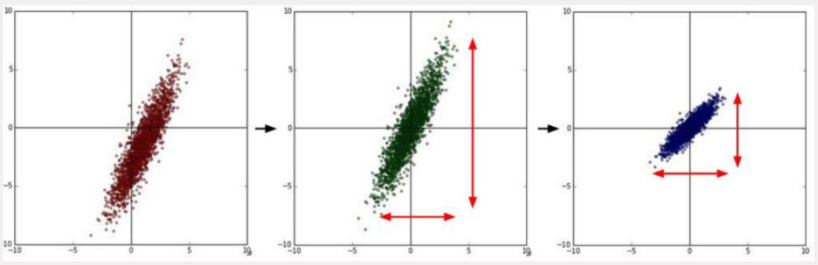
Q: What happens to the gradient when input data is all positive

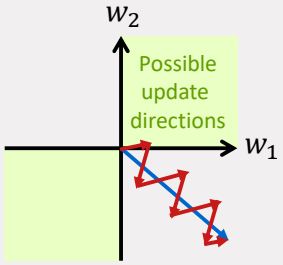
Chain rule

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial w}$$

$$\frac{\partial q}{\partial w_n} = x_n$$

Original data | Zero-mean data | Normalized data





Optimal gradient direction

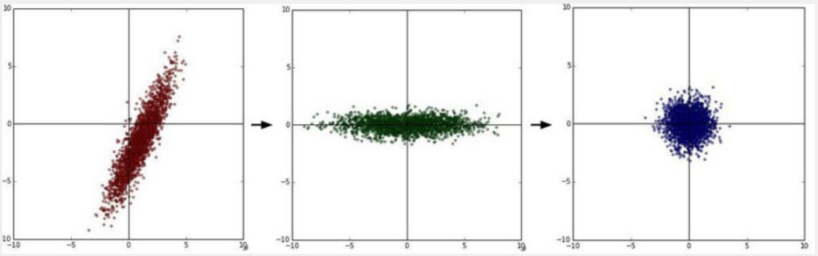
Data normalization

Q: How do we deal with normalization at test time?

Also used:

- Principal Component Analysis (PCA) or whitening

Original data | De-correlated data | Whitened data

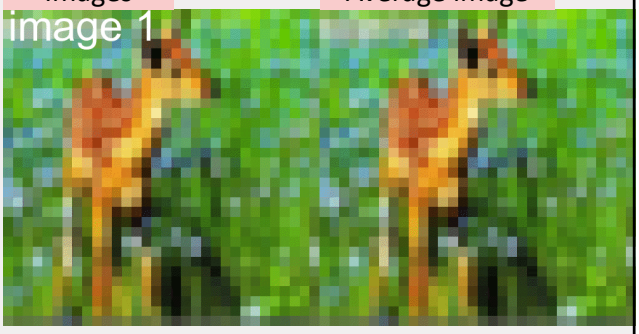


Data normalization

In practice: only make zero-mean

- Subtract mean image
  - *Mean over all images (in the training set!)*
  - *$M \times N \times 3$ for RGB images*
- Subtract mean along channel
  - *Values within a channel quite similar*
  - *1×3 vector (with average R, G and B values)*

Images | Average image



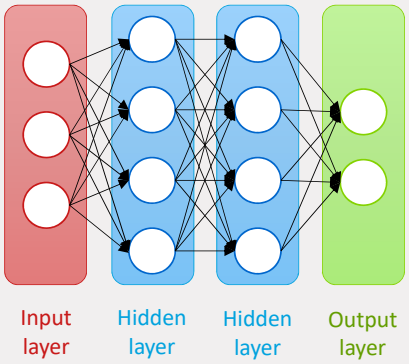
- You don't need to actually do this over all the images
  - *Law of large numbers -> for sufficiently large subset of M samples: $\mu_M \approx \mu_{total}$*

Q: Why not mean over ALL data?

Weight initialization

Before we can start descending in our loss landscape, we need to pick a starting point

- Option 1: start with all-zero weights.
  - *Good idea?*
  - *NO: if all neurons compute the same output, they all have the same influence on the loss, and will follow exactly the same gradient updates.*
- Option 2: Initialize with small random numbers

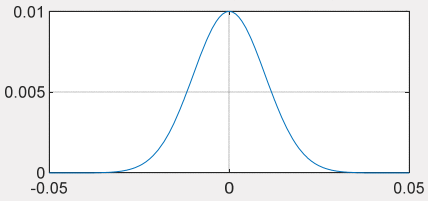


Weight initialization

Initialize with small random numbers

- Sample weights from scaled Gaussian
  - *E.g. zero-mean, 1e-2 std dev.*
- Works OK for small networks…
  - *But problems for deeper networks*
- Example:
  - *10-layer network with 500 neurons in each layer and tanh activation functions*

$$0.01 \cdot \mathcal{N}(0,0.01^2)$$



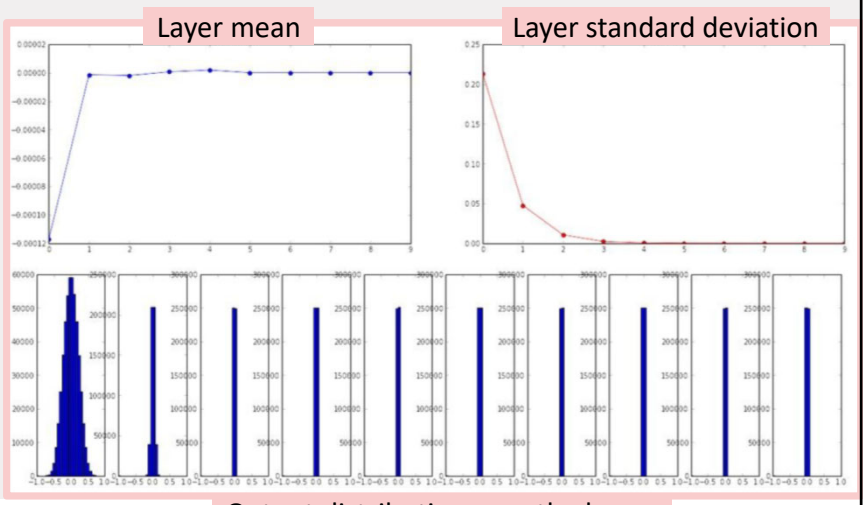
Weight initialization

Observation:

Std dev quickly collapses to zero when we go deeper in the network

What happens during forward pass?

- *Weights small*
- *Input gets multiplied by a small number each layer*
- *Becomes smaller and smaller with each layer*



*Example from cs231n lecture 6 slides 46-51

Weight initialization

Observation:

Std dev quickly collapses to zero when we go deeper in the network

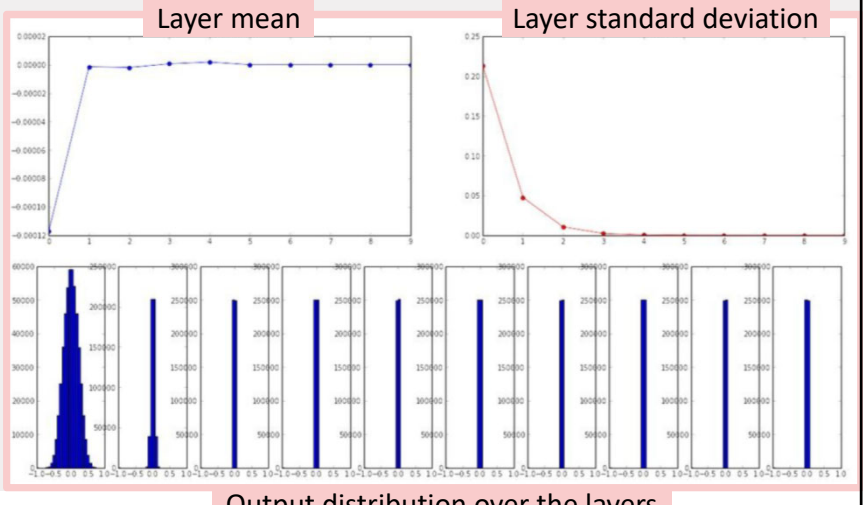
What happens during backward pass?

Weight updates

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial w}$$

Basically no updates!

Will be very small!



*Example from cs231n lecture 6 slides 46-51

Weight initialization

Observation:

Std dev quickly collapses to zero when we go deeper in the network

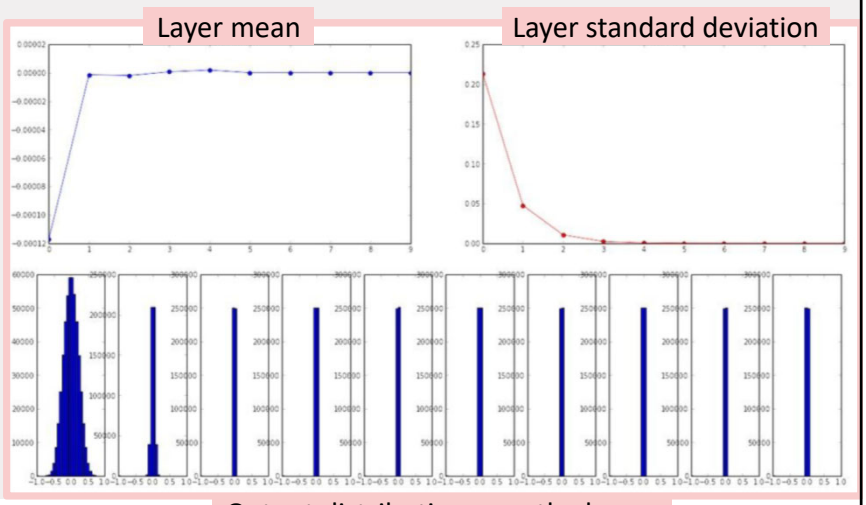
What happens during backward pass?

Backflow gradient

$$\frac{\partial f}{\partial p} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial p}$$

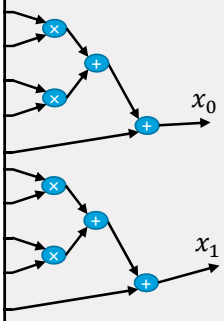
Small backflowing gradient

Will be very small!



Weight initialization

Recall:



Backflowing gradient

For computing the effect of the earlier layers

$$\frac{\partial L}{\partial x_0}$$

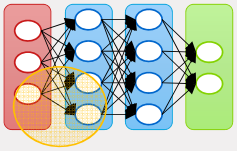
$$\frac{\partial L}{\partial x_1}$$

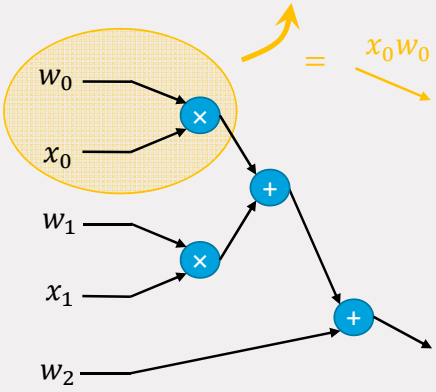
Gradient for weight updates

$$\frac{\partial L}{\partial w_0}$$

$$\frac{\partial L}{\partial w_1}$$

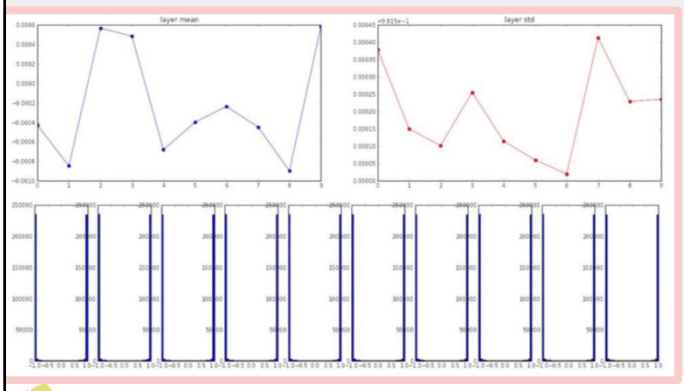
$$\frac{\partial L}{\partial w_2}$$

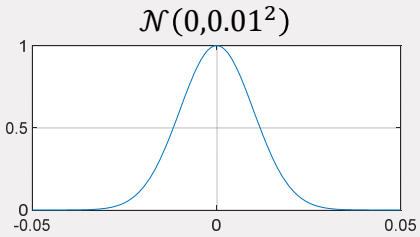




Weight initialization

Option 3: Let's try with big weights!





Observation

- Almost all neurons saturate
  - ➢ *Output either -1 or +1*
- Gradients all become zero!

**Example from cs231n lecture 6 slides 46-51*

Weight initialization

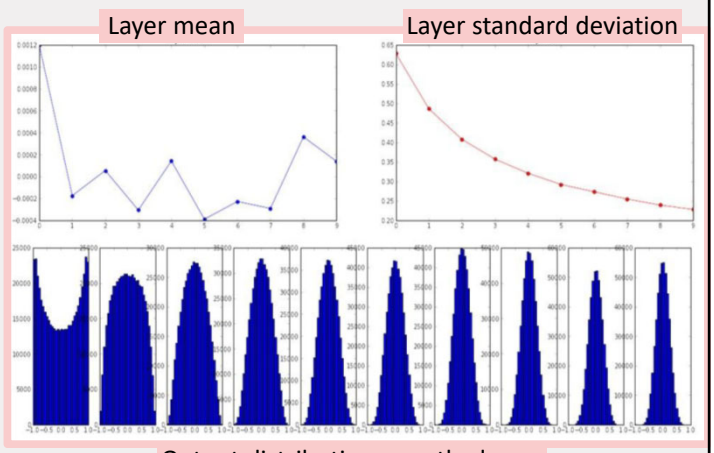
Xavier initialization (Glorot et al., 2010)

Weight initialization is not trivial!

- When they're zero you get inefficient updates…
- When they're too small the layer outputs all collapse…
- When they're too large the layer outputs all saturate…

Reasonable solution:

- Xavier initialization



Output distribution over the layers

**Example from cs231n lecture 6 slides 46-51*

$\mathrm{Var}(XY) = \mathrm{E}(Y)^2\mathrm{Var}(X) + \mathrm{E}(X)^2\mathrm{Var}(Y) + \mathrm{Var}(X)\mathrm{Var}(Y)$

Weight initialization

Neuron output variance scales with the number of inputs

$$\mathrm{Var}(s) = \mathrm{Var}\left(\sum_{i-1}^{N} w_i x_i\right) = \sum_{i-1}^{N} \mathrm{Var}(w_i x_i)$$

$$= \sum_{i-1}^{N} \mathrm{E}(w_i)^2\mathrm{Var}(x_i) + \mathrm{E}(x_i)^2\mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= \sum_{i-1}^{N} \mathrm{Var}(x_i)\mathrm{Var}(w_i) = \mathrm{N} \cdot \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

Weight initialization

Xavier initialization (Glorot et al., 2010)

Scale input x such that

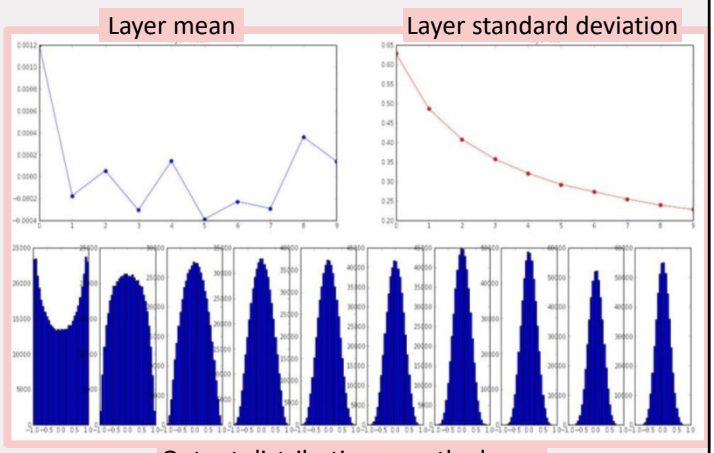
- $\mathrm{Var}(s) = \mathrm{Var}(x)$
- Variance of w should be $1/n$

Since $\mathrm{Var}(\alpha X) = \alpha^2\mathrm{Var}(X)$

➢ Scale x by $\alpha = \sqrt{(1/n)}$

Assumes linear activations

- Active region of tanh



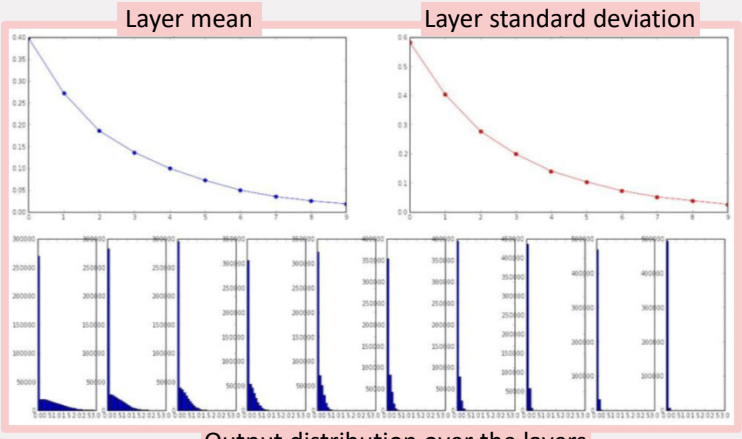
Output distribution over the layers

**Example from cs231n lecture 6 slides 46-51*

Weight initialization

Breaks when using ReLU

- Basically sets half of the units to zero each time
- Outputs collapsing again…



Output distribution over the layers

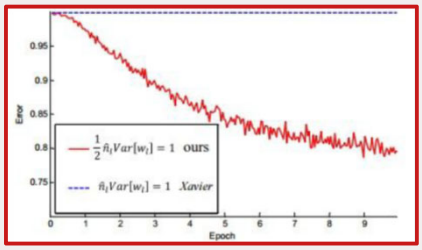
**Example from cs231n lecture 6 slides 46-51*

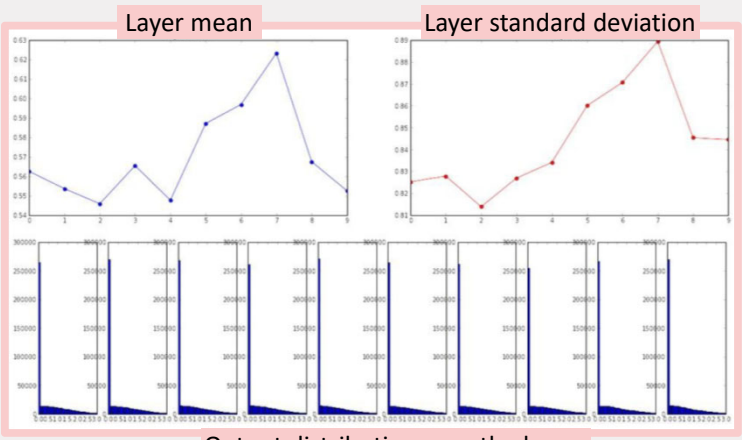
Weight initialization

Solution

- Introduce factor ½

He et al., *"Delving deep into rectifiers: Surpassing human-level performance on imagenet classification"*, CVPR 2015





Output distribution over the layers

**Example from cs231n lecture 6 slides 46-51*

Batch normalization

Needed:

- Unit Gaussian activations at each layer

➢ Let's just make them Gaussian!

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

Consider a batch of activations at some layer

- To make each dimension Guassian, apply:

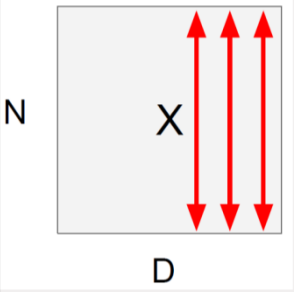
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

→ Differentiable function!

…we can backprop through this!

Batch normalization

Q: Do we really want a unit Gaussian as input to a tanh activation function?

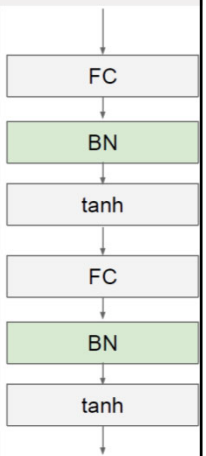


Batch Norm:

1. **Compute the empirical mean and variance independently for each dimension**
2. **Normalize the inputs to "make them Gaussian"**

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Inserted before activation function →



➢ *Note: at test time, use pre-computed mean and std dev over all training data*

**Example from cs231n lecture 6 slides 55-57*

Batch normalization

Squashing and scaling

- Compute the normalized inputs

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- Allow the network to squash the range if it wants to

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

- The networks learns how much BatchNorm it needs for good training!

Note: The network can learn the identity mapping

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

Q: Will a single input always produce the same output during training?

Ready to train?

Pre-process the data

- In practice: zero-mean, variance already constrained by image-type input

Initialize weights

- Avoid vanishing gradients → Xavier / He initialization

Normalize activations

- Batch norm to ensure approximately the same input to each layer

We can start training using mini-batch SGD!

➢ How to monitor training?

➢ How to choose hyper-parameters?

Monitoring the learning process

Before you start: sanity checks

- Double check that the loss is reasonable
  - *Set regularization to zero:*
  - *For cross-entropy loss (and K classes)*

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \underbrace{\lambda R(W)}_{=0}$$

$$L_i = -\log\left(\frac{e^{s_k}}{\sum_j e^{s_j}}\right) \approx \log(K)$$

- Increase regularization → Loss should go up
- Make sure you can overfit a very small portion of the data
  - *(Close to) zero loss, unity accuracy*

Monitoring the learning process

Example:

- First 20 samples of CIFAR-10
- Turn off regularization
- Use vanilla SGD
- Training set = validation set

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
                                    ⋮
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Loss → 0
Accuracy → 1

Monitoring the learning process

Start training!

- Use small regularization factor
- Find an appropriate learning rate
  - *Start with a small learning rate… (e.g. 1e-6)*

Accuracy went up though…

Q: How?

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Cost barely changing…

→ Learning rate too small!

Monitoring the learning process

$$L_i = -\log\left(\frac{e^{s_k}}{\sum_j e^{s_j}}\right) \approx \log(K)$$

Very large scores for other classes

Start training!

- Use small regularization factor
- Find an appropriate learning rate
  - *Start with a small learning rate… (e.g. 1e-6)*
  - *Try a big learning rate… (e.g. 1e+6)*

Not a Number (NaN) → Loss exploded

```
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

Monitoring the learning process

$$L_i = -\log\left(\frac{e^{s_k}}{\sum_j e^{s_j}}\right) \approx \log(K)$$

Very large scores for other classes

Start training!

- Use small regularization factor
- Find an appropriate learning rate
  - *Start with a small learning rate… (e.g. 1e-6)*
  - *Try a big learning rate… (e.g. 1e+6)*
  - *Adjust learning rate down again… (e.g. 1e-3)*

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

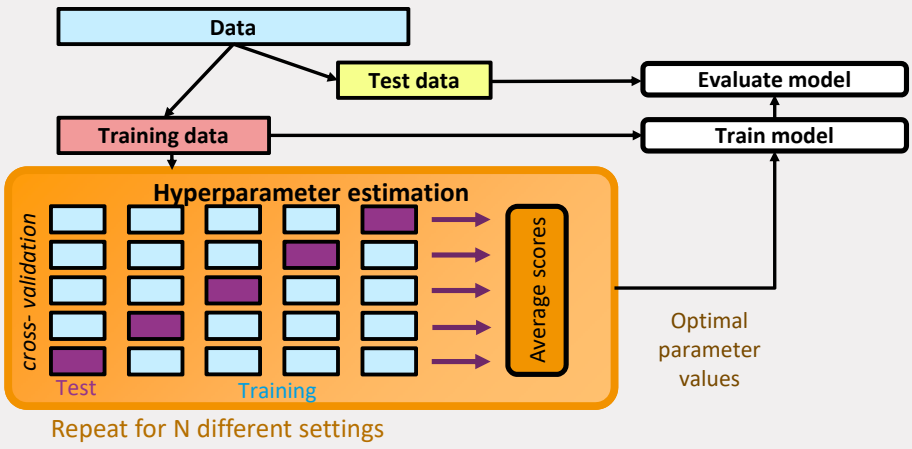
Cost infinity → Loss still exploding

Finding the right learning rate:

- Loss not going down → Learning rate too low
- Loss exploding → Learning rate too high
- Hyper-parameter optimization

Cross-validate [on the training data] for a certain range of different learning rates to find an optimal value.

Hyper-parameter optimization



Hyper-parameter optimization

Finding the parameter range

- Start with a coarse range
  - *Few epochs*
  - *Rough estimate for the optimal values*
  - *Example:*
    - $lr = 10^x, \quad x \in \{-6, -5, \dots, -3\}$
    - $reg = 10^x, x \in \{-5, -4, \dots, +5\}$

Typically sample in log-space

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

Optimal cross-validation results

Hyper-parameter optimization

Finding the parameter range

- Finer grid search in optimal range
  - *Longer running time*
  - *Finer search*
  - *Example:*
    - $lr = 10^x, \quad x \in \{-3 - 4\}$
    - $reg = 10^x, x \in \{-4, -3, \dots, 0\}$

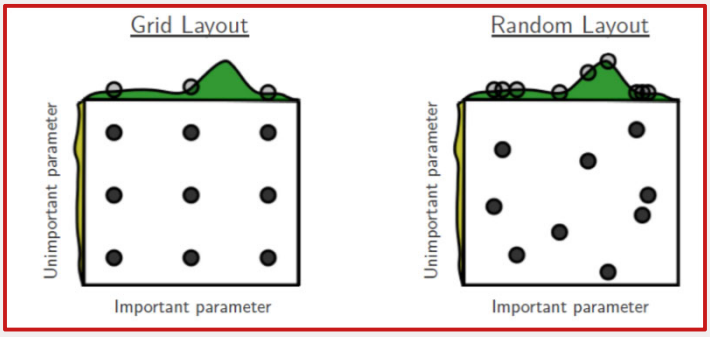
These are just two parameters, there are typically much more values to be set!

Optimal cross-validation results

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

Hyper-parameter optimization

Search range for hyper-parameter optimization



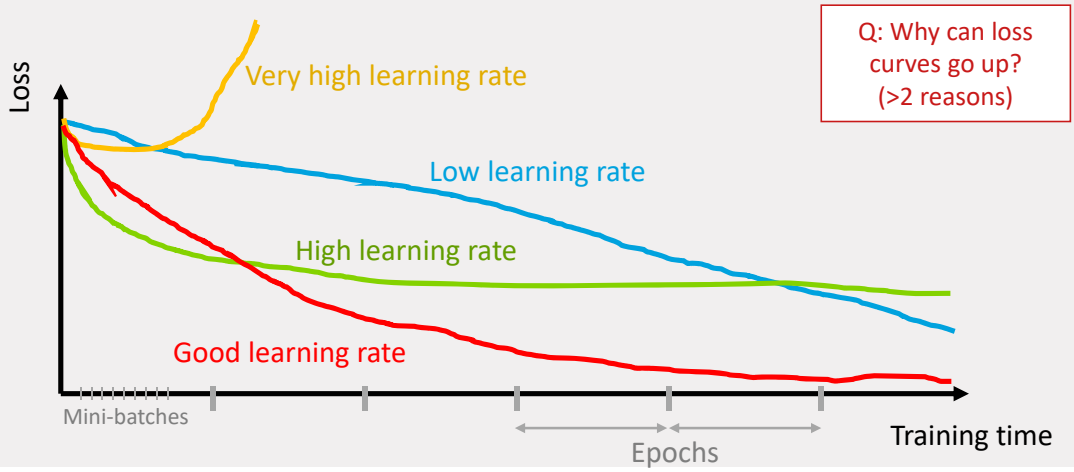
Courtesy Bergstra & Bengio, JMLR, 2012

Reasoning:

- Typically, low correlation between different hyperparameters.
- Hence, random sampling offers more information.

Q: Another reason for random search?

Loss curves



Q: Why can loss curves go up? (>2 reasons)

Summary

Training convnets!

- Can I feed raw data into my network?
  - *You could, but it's better to normalize data!*
- Where should I start in my loss landscape?
  - *Xavier or He initialization to avoid vanishing gradients*
- How can I see if the training goes well?
  - *Check the training and validation loss after each batch/epoch*
- How should I pick my hyperparameters?
  - *Coarse search / fine search, use random sampling across parameter space*
- How can I achieve more robustness against poor initialization?
  - *Batch norm! Normalize before feeding into activation function.*

What's next?

- How can we move down the loss landscape more efficiently?
- Fancy tricks with adjusting the learning rate
- Using multiple models: ensembles
- How should we regularize our model?
- Are there other methods to apply some regularization
- Can we use convnets for (relatively) small data sets?