Course Notes: Deep Learning for Visual Computing

Peter Wonka

August 30, 2021

Contents

1 Network Initialization

1.1 Overview

1.2 Summary / Disclaimer

- **Initialization** of weights is **important**
- **Initialization** of weights is **heuristic**
  - Create a heuristic deriving equations using simplified assumptions
  - Create a heuristic experimentally
- **Different layers** / networks use **different initialization**
- **Initialization in practice**
  - Libraries such as PyTorch or TensorFlow have meaningful default values
  - Use what other SOTA models are using
  - Read the PyTorch documentation TORCH.NN.INIT

1.3 Initialization Considerations

- Initialization depends on four main factors
  - **Number of inputs**: D_{in}
    - e,g. linear layer the dimension of the input vector
    - e,g. conv layer, filter width × filter height × number of input channels
  - **Number of outputs**: D_{out}
  - Type of **non-linearity**: tanh, sigmoid, relu, leaky relu, ...
    - Pytorch uses the concept of $gain$
    - Sigmoid $gain = 1$
    - Tanh $gain = 5/3$
    - ReLU $gain = \sqrt{2}$
    - LeakyReLU $= \sqrt{\frac{2}{1+negativeslope^2}}$

- Type of **network**: RNN, transformer, GAN, ...

1.4 Popular Solutions

- **Uniform** initialization, sample each parameter independently from $U(-a, a)$
  - **Xavier uniform** in PyTorch

$$a = gain \times \sqrt{\frac{6}{D_{in} + D_{out}}} \tag{1.1}$$

  - **Kaiming uniform** in PyTorch

$$a = gain \times \sqrt{\frac{3}{D_{in}}} \tag{1.2}$$

$$a = gain \times \sqrt{\frac{3}{D_{out}}} \tag{1.3}$$

- **Normal** initialization, sample each parameter independently from $N(0, \sigma^2)$

- **Xavier normal** in PyTorch

$$\sigma = gain \times \sqrt{\frac{2}{D_{in} + D_{out}}} \tag{1.4}$$

- **Kaiming normal** in PyTorch

$$\sigma = \frac{gain}{\sqrt{D_{in}}} \tag{1.5}$$

$$\sigma = \frac{gain}{\sqrt{D_{out}}} \tag{1.6}$$

(1.7)

- Traditional **normal**

$$\sigma = \sqrt{\frac{2}{D_{in}}} \tag{1.8}$$

- **Orthogonal** matrix initialization
  - Initialize the weight matrix as orthogonal matrix
- **Truncated Normal** initialization
  - Sample from a truncated normal distribution instead of a regular normal distribution

1.5 Initialization of Biases

- Biases can be initialized with 0
- Alternative initialization: small positive value, e.g. 0.01
  - Idea: stay on the positive side of the ReLU to get more gradient signal
  - Unclear if that works

1.6 Terminology

- Two well known papers
- Literature: Xavier Glorot et al., Understanding the difficulty of training deep feedforward neural networks
  - leads to the name **Xavier** or **Glorot** initialization
- Literature: Kaiming He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
  - leads to the name **Kaiming** or **He** initialization
- **Problem**
  - Terminology is very inconsistent
  - Implementation evolved, but names stay the same

1.7 Potential Problems

- **Symmetry**: different neurons cannot learn different functions
- **Exploding gradients**: gradients in the network become too large
  - Learning is unstable
  - Generation of floating point overflows
- **Vanishing gradients**: gradients in the network become too small
  - Learning is too slow

1.8 Zero Initialization

- Assumption: half the final weights are positive and half the weights are negative
  - Idea: **initialize all weights** with 0 as a best guess estimate
- Problem: neurons (computational outputs) cannot break their **symmetry**
  - All neurons in a layer compute the same output
  - All neurons in the next layer receive the same values as input
  - All neurons will receive the same parameter updates during backpropagation
- Example:
  - Neuron 1: $w_1 x_1 + w_2 x_2$
  - Neuron 2: $w_3 x_1 + w_4 x_2$

1.9 Initializing Network Weights to Small Random Numbers

- Idea: try initializing to approximately zero while breaking symmetry
- Notation: We discuss initialization of a single rank n-tensor $\boldsymbol{W} \in \mathbb{R}^{d_1 \times \ldots \times d_n}$
  - d_i - dimensions along axis i
- **Independent** samples from $N(0,1)$:
  - np.random.randn(d1, ..., dn)
  - Generates $d_1 \times \ldots \times d_n$ scalar values from a Gaussian distribution with 0 mean and variance 1
  - e.g. np.random.randn(5,9) generates a 5×9 matrix of random numbers
- **Independent** samples from $N(\mu, \sigma^2)$
  - sigma*np.random.randn(d1, ..., dn) + mu
  - Notation: sigma, mu in code correspond to σ, μ in equations

- **Note**: multiply with σ not variance σ^2

- **Problem 1**: How small should we make $sigma$, e.g. $sigma = 0.01$?
  - smaller weights results in smaller gradients
- **Problem 2**: Variance of the output depends on the number of inputs

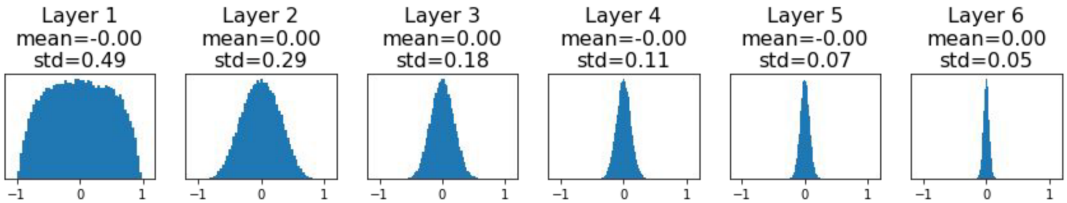
1.10 Experimental Validation

- Idea: easy to experimentally try out different initializations
  - Generate the desired network architecture
  - Initialize the weights according to the proposed algorithm
  - Feed random inputs / training samples to the network
  - Look at the distribution of values in each layer of the network
- First Experiment:
  - tanh activation, 6 linear layers, 4096 values per layer
  - Initialize with small weights:

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

- Results:



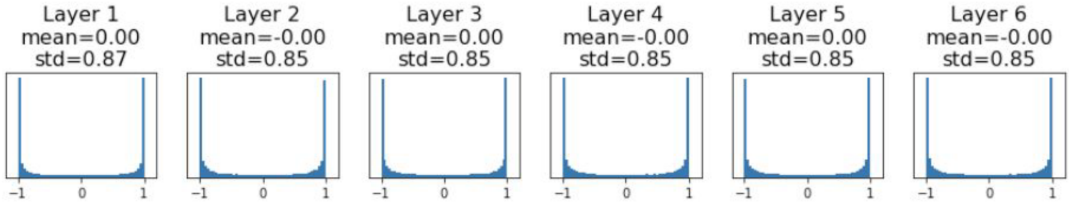
- Problem: output values go towards 0 → gradient goes towards 0

- Second Experiment
  - use larger weights

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

  - Results:



- Problem: output values saturate around $+1$ and $-1 \rightarrow$ gradient goes towards 0

1.11 Variance Review

- Variance of the product of two random variables:

$$\mathrm{Var}(XY) = [\mathbb{E}(X)]^2 \mathrm{Var}(Y) + [\mathbb{E}(Y)]^2 \mathrm{Var}(X) + \mathrm{Var}(X)\mathrm{Var}(Y) \tag{1.9}$$

- Variance of the product of a random variable and a scalar a:

$$\mathrm{Var}(aX) = a^2 \mathrm{Var}(X) \tag{1.10}$$

1.12 Variance Normalization

- Idea: normalize the variance of the output to obtain a nice distribution of values and gradients
  - empirically improves convergence
  - based on analyzing a network without non-linear activation functions
- Code: `w = np.random.randn( ... ) * sqrt(1/D_in)`
- Simplified derivation:
  - Assumptions:
    - x_i, w_i are zero mean (not true for ReLU)
    - $\mathbb{E}(w_i) = \mathbb{E}(x_i) = 0$
    - x_i, w_i are iid
    - x_i, w_i are independent

- weights w_i, inputs x_i

$$s = \sum_{i}^{D_{in}} w_i x_i \tag{1.11}$$

$$\mathrm{Var}(s) = \mathrm{Var}\left(\sum_i w_i x_i\right) \tag{1.12}$$

$$= \sum_i \mathrm{Var}(w_i x_i) \tag{1.13}$$

$$= \sum_i [\mathbb{E}(w_i)]^2 \mathrm{Var}(x_i) + [\mathbb{E}(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \tag{1.14}$$

$$= \sum_i \mathrm{Var}(x_i)\mathrm{Var}(w_i) \tag{1.15}$$

$$= D_{in}\mathrm{Var}(w)\mathrm{Var}(x) \tag{1.16}$$

$$\tag{1.17}$$

- Goal: make the variance of the inputs x_i the same as the variance of the output s
  - variance of the weights should be $1/D_{in}$
  - Property of variance: $\mathrm{Var}(aX) = a^2 \mathrm{Var}(X)$
  - Therefore, scale weights by $\sqrt{\frac{1}{D_{in}}}$
- Intuition: ReLU sets about half the values to 0 → variance decreases by $1/2$
  - scale weights by $\sqrt{\frac{2}{D_{in}}}$

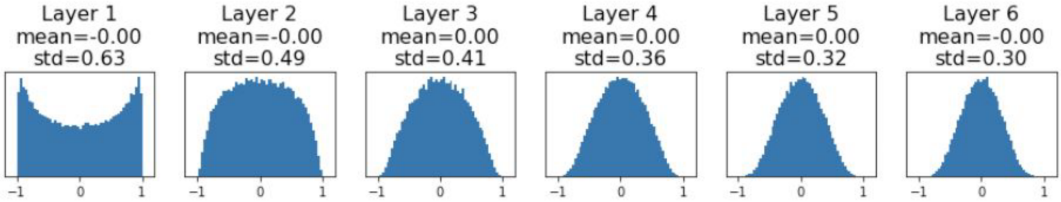
1.13 Experimental Validation

- First Experiment:

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

  - Results:



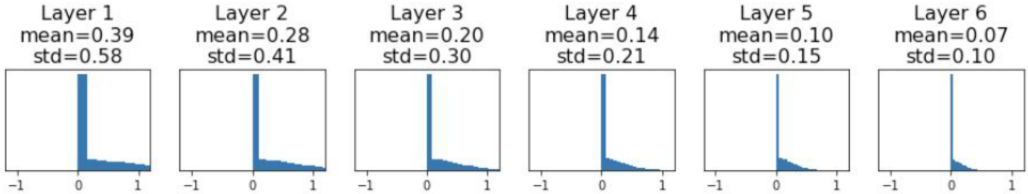
- Analysis: looks good

- Second Experiment
  - switch from tanh to ReLU

Change from tanh to ReLU

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

- Results:

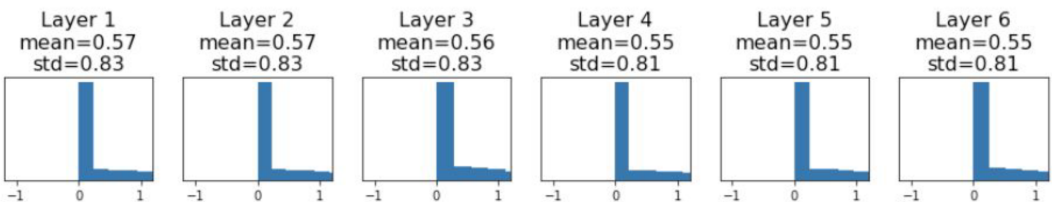


- Problem: output values go towards 0 → gradient goes towards 0

1.14 Experimental Validation

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)



- Analysis: looks good

1.15 Interaction with other Components

- Initialization interacts with other components
- Interaction with normalization layers seems significant
  - e.g. Batch normalization can normalize before the non-linear activation function to change the variance and mean.