THEORY

Feed-forward and convolutional neural networks
A bit of History
**A bit of History**

**McCulloch & Pitts / Hebb**

Rosenblatt’s Perceptron

Minsky & Papert - XOR

Backpropagation Algorithm

- **1943 Warren McCulloch & Walter Pitts:**
  - How To: From neurons to complex thought
  - Binary threshold activations

- **1949 Howard Hebb:**
  - Neurons that fire together wire together
  - Weights: Learning and memory

[McCulloch, 43]
1948, Rosenblatt applied Hebb’s learning to McCulloch & Pitts design

\[
f(x) = \begin{cases} 
1 & \text{if } w \cdot x + b > 0 \\
0 & \text{otherwise}
\end{cases}
\]

- \( w \) real-valued weights
- \( \cdot \) dot product
- \( b \) real scalar constant

The Mark I Perceptron. A visual classifier with:

- 400 photosensitive receptors (sensory units)
- 512 stepping motors (association units, trainable)
- 8 output neurons (response units)
Rosenblatt acknowledged a set of limitations in the Perceptron machine.

Minsky & Papert did too in “Perceptrons: an introduction to computational geometry”, including:
- A multilayer perceptron (MLP) is needed for learning basic functions like XOR
- MLP cannot be trained.

This had a huge impact on the public, resulting in a drastic cut in funding of NNs until the mid 80s

1st AI WINTER

[Minsky, 69]
A bit of History

McCulloch & Pitts / Hebb
Rosenblatt's Perceptron
Minsky & Papert - XOR
Backpropagation Algorithm

How can we optimize neuron weights which are not directly connected to the error measure?

**Backpropagation** algorithm:
*Use the chain rule to find the derivative of cost with respect to any variable.*

In other words, find the contribution of each weight to the overall error.

First proposed for training MLPs by *Werbos* in '74.
Rediscovered by *Rumelhart, Hinton and Williams* in '85.

**End of NNs Winter**

Training with backprop
1. Forward pass from input to output
2. Error measurement (loss function)
3. Find gradients towards minimizing error layer by layer (backward pass)
Feedforward Neural Networks
Computing the gradients using all available training data would require huge amounts of memory.

**Stochastic Gradient Descent**: Iteratively update weights using random samples (hence, *stochastic*).

Each feedforward/backward cycle (a **step**) processes a random **batch** of images.
- Typical batch sizes: Powers of 2.
- Batch size = 1 --> Full stochastic (slower)
- Batch size = dataset_size --> Deterministic (bad generalization)

An **epoch** is the processing of the whole dataset once. It corresponds to processing as many batches as:

\[ \text{dataset}_\text{size} / \text{batch}_\text{size} \]
Activation functions transform the output of a layer to a given range. If the function is non-linear, the net can learn non-linear patterns (e.g., XOR).

- Zero gradient in most of f(x). Saturates!
- Max gradient is 0.25 or 1. Vanishing!
- Does not saturate
- Does not vanish
- Faster
- May die

ReLU is a safe choice in most cases
Undying alternatives: Leaky ReLU, ELU, ...
Gradient descent is a simple and straight-forward optimization algorithm to update weights towards a min.

**Learning rate** determines how much we move in that direction. With the wrong LR you may end up in local minima or saddle points, or be too slow.

SGD will overshoot unless we keep decreasing the LR.
**Momentum**: Include a fraction of the previous gradient. Keeps the general direction so far.

**Nesterov**: Compute current gradient considering where the previous gradient took you. (RNNs?)

**Adagrad**: Parameter-wise LR considering past updates. Good for infrequent patterns (GloVe). Vanishing LR due to growing history.

**Adadelta**: Adagrad with a decaying average over history. Typically set around 0.9.

**Adam**: Adadelta + Momentum
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Why do we need regularization?

Because the difference between Machine Learning and Optimization is called Generalization
Feedforward Neural Networks

Generalization

1. \( h(x) = w_1 x + b \)
2. \( h(x) = w_3 x^3 + w_2 x^2 + w_1 x + b \)
3. \( h(x) = w_{14} x^{14} + w_{13} x^{13} + \cdots + w_1 x + b \)

Polynomial regression

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activations functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Generalization

Polynomial regression

<table>
<thead>
<tr>
<th>Training Error</th>
<th>Huge</th>
<th>Small</th>
<th>Tiny</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Model Generalization</th>
<th>Bad</th>
<th>Good</th>
<th>Horrible</th>
</tr>
</thead>
</table>

![Graphs showing training error and model generalization](image)
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Generalization

What policy can we use to improve model generalization?

Occam’s Razor
when you have two competing hypotheses
that make the same predictions,
the simpler one is the better

Machine Learning
given two models
that have a similar performance,
It’s better to choose the simpler one
Feedforward Neural Networks

Model Complexity

What policy can we use to improve model generalization?

Cost function = Training Error + Model Complexity
Feedforward Neural Networks

Model Complexity

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \quad \text{vs} \quad h(x) = 0x^3 + 0x^2 + w_1 x + w_0 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Model Complexity

\[ h(x) = w_3x^3 + w_2x^2 + w_1x + w_0 \quad \text{VS} \quad h(x) = 0x^3 + 0x^2 + w_1x + w_0 \]

\[ h(x) = w_3x^3 + w_2x^2 + w_1x + w_0 \quad \text{VS} \quad h(x) = 0x^3 + w_2x^2 + 0x + w_0 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Model Complexity

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \]

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0 \]

\[ h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Model Complexity

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \]

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0 \]

\[ h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0 \]

\[ h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \quad \text{VS} \quad h(x) = 0 x^3 + w_2 x^2 + 0 x + 0 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Model Complexity

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \]  \text{VS}  \quad \color{red}{h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0} 

\[ h(x) = w_3 x^3 + w_2 x^2 + w_1 x + w_0 \]  \text{VS}  \quad \color{red}{h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0} 

\[ h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \]  \text{VS}  \quad \color{red}{h(x) = 0 x^3 + w_2 x^2 + 0 x + w_0} 

\[ h(x) = 0 x^3 + 0 x^2 + w_1 x + w_0 \]  \text{VS}  \quad \color{red}{h(x) = 0 x^3 + w_2 x^2 + 0 x + 0}
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

Model Complexity

\[ h(x) = 0x^3 + 0x^2 + w_1x + w_0 \quad \text{VS} \quad h(x) = 0x^3 + w_2x^2 + 0x + 0 \]

- \( \ell_0 \) complexity: Number of non-zero coefficients
- \( \ell_1 \) "lasso" complexity: \( \sum_{i=0}^{d} |w_i| \), for coefficients \( w_0, ..., w_d \)
- \( \ell_2 \) "ridge" complexity: \( \sum_{i=0}^{d} w_i^2 \), for coefficients \( w_0, ..., w_d \)
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Model Complexity

\[ h(x) = 0x^3 + 0x^2 + w_1x + w_0 \quad \text{vs} \quad h(x) = 0x^3 + w_2x^2 + 0x + 0 \]

\[ w_0 = 1.3 \quad w_1 = -1.2 \quad \text{vs} \quad w_2 = 2.2 \]

\( \ell_0 \) complexity

\[ |\{w_1, w_0\}| = 2 \quad \text{vs} \quad |\{w_2\}| = 1 \]

\( \ell_1 \) complexity

\[ |1.3| + |-1.2| = 2.5 \quad \text{vs} \quad |2.2| = 2.2 \]

\( \ell_2 \) complexity

\[ 1.3^2 + (-1.2)^2 = 3.13 \quad \text{vs} \quad 2.2^2 = 4.84 \]
L1 / L2 Regularization

Cost function = Loss + \( \frac{\lambda}{m} \sum_{i=0}^{m} |w_i| \)

Cost function = Loss + \( \frac{\lambda}{2m} \sum_{i=0}^{m} w_i^2 \)

Regularization parameter \( \rightarrow \lambda \)

What **complexities** do these methods use?

\(\ell_1\) ”lasso” complexity: \( \sum_{i=0}^{d} |w_i| \), for coefficients \( w_0, ..., w_d \)

\(\ell_2\) ”ridge” complexity: \( \sum_{i=0}^{d} w_i^2 \), for coefficients \( w_0, ..., w_d \)
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

L1 / L2 Regularization

L1 Penalty

\[ |w| \]

L2 Penalty

\[ w^2 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

L1 / L2 Regularization

\[ |w| \]

\[ w^2 \]

\[ \Delta w \]

\[ \Delta w \]

\[ \Delta \mathcal{L} \]

\[ \Delta \mathcal{L} \]

L1 Penalty

L2 Penalty
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Dropout

Step n

Dropout 50%

Dropout 50%
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Dropout

Step n+1
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Dropout
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

**Dropout**

Before drop-out:

\[ a_0^{[0]} = g \left( w_{00}^{[0]} x_0 + w_{10}^{[0]} x_1 + w_{20}^{[0]} x_2 + w_{30}^{[0]} x_3 + w_{40}^{[0]} x_4 + b_0^{[0]} \right) \]

After drop-out: \[ a_0^{[0]} = 0 \]

What **complexity** does this method use?

\( \ell_0 \) complexity: Number of non-zero coefficients
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

Early Stopping

![Graph showing Early Stopping](image)

- Error
- Validation
- Training
- Stop training
- Number of epochs
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Early Stopping

![Early Stopping Diagram](image)

- Error
- Validation
- Training
- Stop training
- Number of epochs
Early Stopping

What **complexity** does this method use?

\[ \ell_1 \text{ "lasso" complexity: } \sum_{i=0}^{d} |w_i|, \text{ for coefficients } w_0, \ldots, w_d \]

\[ \ell_2 \text{ "ridge" complexity: } \sum_{i=0}^{d} w_i^2, \text{ for coefficients } w_0, \ldots, w_d \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ x = \frac{x - \mu}{\sigma^2} \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
**Normalizing inputs**
Vanishing/Exploding Gradients
Weights initialization

\[ x = \frac{x - \mu}{\sigma^2} \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ x = \frac{x - \mu}{\sigma^2} \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Why input normalization matters?

Unnormalized loss space

$\begin{align*}
    f_1 &\in [1, 1000] \\
    f_2 &\in [0, 1]
\end{align*}$

Normalized loss space

$\begin{align*}
    f_1 &\in [-0.5, 0.5] \\
    f_2 &\in [-0.5, 0.5]
\end{align*}$
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Why input normalization matters?

Unnormalized loss space

\[ f_1 \in [1, 1000] \]
\[ f_2 \in [0, 1] \]

Normalized loss space

\[ f_1 \in [-0.5, 0.5] \]
\[ f_2 \in [-0.5, 0.5] \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

\[
\begin{align*}
\nabla_X J & \\
\nabla W_0 J & \\
\nabla W^1 J & \\
\nabla W^2 J & \\
\nabla W^3 J & \\
\end{align*}
\]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[
\nabla_x J \quad \nabla_{W^0} J \quad \nabla_{W^1} J \quad \nabla_{W^2} J \quad \nabla_{W^3} J
\]
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

\[ \nabla W^0 L \]

Weights get stucked!
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Why \textit{gradients} get \textit{smaller and smaller} at each layer on backpropagation?
Why gradients get smaller and smaller at each layer on backpropagation?

Well, this is related to the derivatives chain rule.
Why gradients get smaller and smaller at each layer on backpropagation?

\[ \nabla_w^{L-2} J = \frac{\partial J}{\partial w_{21}^L} = \frac{\partial J}{\partial a_1^L} \times \frac{\partial a_1^L}{\partial z_1^L} \times \frac{\partial z_1^L}{\partial w_{21}^L} \]

\[ \nabla_w^{L-1} J = a \times b \times c \]

\[ \nabla_w^{L} J = a \times b \times c \times d \times e \times f \]

\[ \nabla_w^{L-2} J = a \times b \times c \times d \times e \times f \times g \ldots \]
Why \textbf{gradients} get \textbf{smaller and smaller} at each layer on backpropagation?

\[ \nabla_{w_{ij}^L} J = a \times b \times c \]
\[ \nabla_{w_{ij}^{L-1}} J = a \times b \times c \times d \times e \times f \]
\[ \nabla_{w_{ij}^{L-2}} J = a \times b \times c \times d \times e \times f \times g \ldots \]

Terms values < 1.0

\[ \nabla_{W^0} J \quad \nabla_{W^1} J \quad \nabla_{W^2} J \quad \nabla_{W^3} J \]

\[ \downarrow \downarrow \downarrow \quad \downarrow \downarrow \quad \downarrow \quad \downarrow \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

What about **exploding gradients**?

\[
\nabla_{w_{ij}}^L J = a \times b \times c \\
\nabla_{w_{ij}}^{L-1} J = a \times b \times c \times d \times e \times f \\
\nabla_{w_{ij}}^{L-2} J = a \times b \times c \times d \times e \times f \times g \ldots
\]

Terms values > 1.0

\[\nabla W^0 J \quad \nabla W^1 J \quad \nabla W^2 J \quad \nabla W^3 J \]

\[\uparrow \uparrow \uparrow \uparrow \quad \uparrow \uparrow \quad \uparrow \quad \uparrow \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Normal Distribution $N(\mu = 0, \sigma^2 = 1)$

Feedforward Neural Networks

$W^l_{ij}$

$a_{0}^{l-1}$
$a_{1}^{l-1}$
$a_{2}^{l-1}$
$a_{3}^{l-1}$
$a_{N}^{l-1}$

$Z_i^l$

$a_{0}^{l-1}$: 0.43
$a_{1}^{l-1}$: -1.83
$a_{2}^{l-1}$: -0.37
$a_{3}^{l-1}$: -0.11

$N = 200$
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ N = 200 \]

Normal Distribution \( N(\mu = 0, \sigma^2 = 1) \)
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[
X \sim N(\mu_X, \sigma^2_X) \\
Y \sim N(\mu_Y, \sigma^2_Y) \\
Z = X + Y \\
Z \sim N(\mu_X + \mu_Y, \sigma^2_X + \sigma^2_Y)
\]

Sum of independent random variables that are normally distributed:
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Sum of independent random variables that are normally distributed:

\[ X \sim N(\mu_X, \sigma^2_X) \]
\[ Y \sim N(\mu_Y, \sigma^2_Y) \]
\[ Z = X + Y \]

\[ Z \sim N(\mu_X + \mu_Y, \sigma^2_X + \sigma^2_Y) \]

\[ \mu(Z_i^l) = 0.0 \]
\[ \sigma(Z_i^l) = \sqrt{200} \approx 14.14 \]

\[ N = 200 \]
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

\[
sigmoid(x) = \frac{e^x}{e^x + 1}
\]

\[
\sigma(z^l_i) = \sqrt{200} \approx 14.14
\]

\[
\mu(z^l_i) = 0.0
\]

\[
N = 200
\]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Alternative weights initializations?

\[
W^l \cdot \mathbf{z}_i
\]

\[N = 200\]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Normal Distribution
\( N(\mu = 0, \sigma^2 = 1) \)

\[
\begin{align*}
&W_{13}^l \\
&\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
1
\end{array} \\
&\begin{array}{c}
???
\\
???
\\
???
\end{array} \\
&\begin{array}{c}
Z_i^l
\end{array} \\
&N = 200
\end{align*}
\]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ N = 200 \]

\[ N(\mu = 0, \sigma^2 = 1) \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ W^l_{i,3} \]

\[ Z^l_i \]

\[ N = 200 \]

\[ N(\mu = 0, \sigma^2 = 1) \]

Truncated Normal Initialization

Normal Distribution

\[ N(\mu = 0, \sigma^2 = 1) \]

\[ -4 \quad -2 \quad 0 \quad 2 \quad 4 \]

\[ 0.4 \quad 0 \quad 0.4 \]
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

$N = 200$

Truncated Normal Initialization

Normal Distribution $N(\mu = 0, \sigma^2 = 1)$
Feedforward Neural Networks

- SGD, Epochs, Batches and Steps
- Activation functions
- SGD learning rate
- Other optimization methods
- Regularization
- Normalizing inputs
- Vanishing/Exploding Gradients
- Weights initialization

Truncated Normal Initialization

Normal Distribution $N(\mu = 0, \sigma^2 = 1)$

Feedforward Neural Network diagram

$W^l_{13}$

$Z^l_i$

$N = 200$
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

Xavier / Glorot Initialization

$\mu(z_i^l) = 0.0$
$\sigma(z_i^l) = 1.0$

$\sigma^2(W.3_i) = \frac{1}{N} = 0.005$
$\sigma(W.3_i) = \sqrt{\frac{1}{N}} \approx 0.0707$

$N = 200$
Feedforward Neural Networks

SGD, Epochs, Batches and Steps
Activation functions
SGD learning rate
Other optimization methods
Regularization
Normalizing inputs
Vanishing/Exploding Gradients
Weights initialization

\[ W_i^l \cdot \mathbf{x}_i \]

\[ Z_i^l \]

\[ N = 200 \]

Xavier / Glorot Initialization

Normal Distribution
\[ N(\mu = 0, \sigma^2 = 0.005) \]
Practical Aspects
Train / Validation / Test Workflow

- **Train** Set used to **train** & **control bias**
- **Validation** Set used to **control variance**
- **Test** Set used to **estimate the generalization error** of the final model
Train / Validation / Test

Dataset

Train Set used to train & control bias

Validation Set used to control variance

Test Set used to estimate the generalization error of the final model
Dataset

Train

Train Set used to **train & control bias**

Test

Test Set used to **control variance**

Set used to **estimate the generalization error** of the final model

Practical Aspects

Train / Validation / Test Workflow
Practical Aspects

Train / Validation / Test Workflow

Machine Learning

- Train: As many as possible
- Validation: The minimum amount to appropriately represent each class
- Test: The minimum amount to appropriately represent each class

How many images do we need for each set?

- Train: 80%
- Validation: 10%
- Test: 10%
Practical Aspects

Train / Validation / Test

Workflow

Machine Learning

80%

10%

10%

Typical dataset size: 1.000 - 30.000

Deep Learning

99%

Typical dataset size: 30.000 - 10.000.000
Practical Aspects

Train / Validation / Test Workflow
Practical Aspects

Train / Validation / Test

Workflow

Get Data
2
Train Model
3
Clean, Prepare & Manipulate Data

Test Data
4
Lots of hyper-parameters:
• Network architecture:
  • # layers
  • # neurons
  • Activation function
• Learning rate
• Optimization algorithm
• ...
Practical Aspects

Train / Validation / Test Workflow

Reduce high bias

- Train longer
- Increase model’s capacity:
  - More layers
  - More hidden neurons
- Search for architecture suitable for your data structure
- Preprocessing data to simplify the task

Reduce high variance

- More data
- Data Augmentation
- Regularization
- Search for architecture suitable for your data structure
Convolutional Neural Networks
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside
CNN Applications

Some data has spatial correlations that could be exploited (in 1D, 2D, 3D, ...):
• Near-by data points are more relevant than far-away.

If we sparsify connectivity with a consistent purpose, we may reduce complexity and ease the learning of more coherent patterns.
Sparse connectivity is nice, but we still want to apply filters everywhere.

Each limited connectivity pattern (a **kernel**) will get **convolved** all over the image, generating a number of values. Notice each kernel generates a 2D matrix of values.

In practice we have sets of neurons **sharing** weights.
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside
CNN Applications

Convolution kernels can do all sorts of things on an image:

Let's let the model learn them
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
**Kernel size, stride and padding**
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside
CNN Applications

**Kernel size**: Size of the receptive field of convolutional neurons. Typically 3x3, 5x5, 7x7

**Stride**: Number of steps while convolving filter.

![Diagram showing convolutional layers and kernel sizes](image)

Stride 1 the most common. Larger strides can replace pooling.

**Padding**: Border added to center conv. everywhere
- No padding: Dimensionality reduced
- Most common, zero equal/same padding

\[
\text{OutputSize} = \frac{\text{InputSize} - \text{KernelSize} + 2 \times \text{Padding}}{\text{Stride}} + 1
\]
Convolutional Neural Networks

- Limited connectivity
- Convolution & weight sharing
- Filters
- Kernel size, stride and padding

**Convolutional volumes**

- Pooling layers
- Convolutional architectures
- CNNs from the inside
- CNN Applications

- In a typical 2D CNN, conv filters are 3D (full depth).
- Each filter convolved generates a 2D plane of data.
- Depth provides all the neural views on a part of data.
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside
CNN Applications

Pooling:
- Small spatial invariance
- Dimensionality reduction (along x and y only)
- Never applied full depth!
- Parameter free layer
- Hyperparams:
  - Size & Stride
- Loss in precision
- Max >> Avg

Pooling layers:
- Small spatial invariance
- Dimensionality reduction (along x and y only)
- Never applied full depth!
- Parameter free layer
- Hyperparams:
  - Size & Stride
- Loss in precision
- Max >> Avg
The first influential architecture was **AlexNet**:
- 5 layers using convs, pools, ReLU, 2 dense, and dropout.
- 62M parameters

**VGG16/19** extends the (conv-pool)*dense design:
- Smaller, 3x3 filters, but more
- 138M parameters

Some design principles: KISS, be repetitive & pyramidal

Bigger is not better!
Convolutional Neural Networks

- Limited connectivity
- Convolution & weight sharing
- Filters
- Kernel size, stride and padding
- Convolutional volumes
- Pooling layers
- Convolutional architectures
- CNNs from the inside
- CNN Applications

But deeper should never be worse!
  - In theory, yes. In practice, identity is hard to learn

**ResNet:** Learning zero is easier than learning id.
  - We can now train a 1K layer net

**DenseNet:** link all to all
  - Use depth concats
  - 1x1 convs to make it feasible

**Inception:** how to fix filter size?
  - Let the net decide which is best
  - Avg. Pooling instead of dense
Different architectures that can be done...

- Convolution – Transposed convolution (pixel-wise)
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures

CNNs from the inside

CNN Applications

What do filters learn?
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside
CNN Applications

Style transfer

Multimodal pipelines
Convolutional Neural Networks

Limited connectivity
Convolution & weight sharing
Filters
Kernel size, stride and padding
Convolutional volumes
Pooling layers
Convolutional architectures
CNNs from the inside

CNN Applications

[Zhang,16]
[Iizuka,16]
[Chen,16]