DeepLearning on FPGAs
Artificial Neural Networks: Backpropagation and more

Sebastian Buschhäuser

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

November 10, 2016
Recap: Homework

**Question:** So what's your accuracy?
**Question:** What about speed?
Recap: Homework

**Question:** So what's your accuracy?

**Question:** What about speed?

**Some remark about notation:** In the previous slides I used $\theta$ twice with different meaning

1) As “bias” parameter for the perceptron
2) As vector-to-be-optimized by gradient descent

$\Rightarrow$ This is now changed. $\theta$ will always be used in a general fashion as the vector-to-be-optimized.

**Any questions / remarks / whatsoever?**
Recap: Data Mining (1)

Important concepts:

- **Feature Engineering** is key to solve Data Mining tasks
- **Deep Learning** combines learning and Feature Engineering

Data Mining approach:

- Specify model family (→ perceptron)
- Specify optimization procedure (→ gradient descent)
- Specify a cost / loss function (→ RMSE or cross-entropy)
Recap: Data Mining (1)

Important concepts:

- **Feature Engineering** is key to solve Data Mining tasks
- **Deep Learning** combines learning and Feature Engineering

Data Mining approach:

- Specify model family (→ perceptron)
- Specify optimization procedure (→ gradient descent)
- Specify a cost / loss function (→ RMSE or cross-entropy)

**Perceptron:** A linear classifier \( f: \mathbb{R}^d \to \{0, 1\} \) with

\[
\hat{f}(\bar{x}) = \begin{cases} 
+1 & \text{if } \sum_{i=1}^{d} w_i \cdot x_i \geq b \\
0 & \text{else}
\end{cases}
\]
Recap: Data Mining (2)

Optimization procedure: Gradient descent

\[ \hat{\theta}^{\text{new}} = \hat{\theta}^{\text{old}} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{\text{old}}) \]
Recap: Data Mining (2)

Optimization procedure: Gradient descent

\[
\hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old})
\]

Loss function: RMSE or cross-entropy

\[
\ell(D, \hat{\theta}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - f_{\hat{\theta}}(\vec{x}_i))^2}
\]

\[
\ell(D, \hat{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} (y_i \ln (f_{\hat{\theta}}(\vec{x}_i)) + (1 - y_i) \ln (1 - f_{\hat{\theta}}(\vec{x}_i)))
\]
Recap: Data Mining (2)

Optimization procedure: Gradient descent

$$\hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old})$$

Loss function: RMSE or cross-entropy

$$\ell(D, \hat{\theta}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - f_{\hat{\theta}}(\vec{x}_i))^2}$$

$$\ell(D, \hat{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} (y_i \ln(f_{\hat{\theta}}(\vec{x}_i)) + (1 - y_i) \ln(1 - f_{\hat{\theta}}(\vec{x}_i)))$$

So far: Training of single perceptron
Now: Training of multi-layer perceptron (MLP)
**MLP: Some Notation (1)**

\[
\text{We can represent the MLP model as follows:}
\]

\[
\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_d \rightarrow \mathbf{w}_{ij}^{(l+1)} \rightarrow \mathbf{j} \rightarrow \mathbf{\hat{y}}
\]

\[
\hat{y} = \mathsf{f}(i)_{(l+1)}
\]

\[
\mathbf{w}_{ij}^{(l+1)} \equiv \text{Weight from neuron } i \text{ in layer } l \text{ to neuron } j \text{ in layer } l + 1
\]
MLP: Learning

**Obviously:** We need to learn the weights $w^{(l)}_{i,j}$ and bias $b^{(l)}_j$

**So far:** We intuitively derived a learning algorithm

\[
\hat{w}^{new} = \hat{w}^{old} - \alpha \cdot \nabla \hat{w} \ell(D, \hat{w})
\]

**Loss function:**

\[
\ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\vec{x}_i))^2}
\]
MLP: Learning

Obviously: We need to learn the weights $w_{i,j}^{(l)}$ and bias $b_j^{(l)}$

So far: We intuitively derived a learning algorithm

Observation: For MLPs we can compare the output layer with our desired output, but what about hidden layers?

Thus: We use gradient descent + “simple” math
MLP: Learning

Obviously: We need to learn the weights $w_{i,j}^{(l)}$ and bias $b_j^{(l)}$

So far: We intuitively derived a learning algorithm

Observation: For MLPs we can compare the output layer with our desired output, but what about hidden layers?

Thus: We use gradient descent + “simple” math

Gradient descent:

$$\hat{w}^{new} = \hat{w}^{old} - \alpha \cdot \nabla_{\hat{w}} \ell(D, \hat{w})$$

Loss function:

$$\ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left( y_i - \hat{f}(\vec{x}_i) \right)^2}$$
MLP: Learning (2)

\[ \ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\vec{x}_i))^2} \]

**Observation:** We need to take the derivative of the loss function.
MLP: Learning (2)

\[ \ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\vec{x}_i))^2} \]

**Observation**: We need to take the derivative of the loss function

**But**: Loss functions looks complicated

**Observation 1**: Square-Root is monotone

**Observation 2**: Loss function depends on entire training data set!
MLP: Learning (2)

\[ \ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\vec{x}_i))^2} \]

**Observation:** We need to take the derivative of the loss function

**But:** Loss functions looks complicated

**Observation 1:** Square-Root is monotone

**Observation 2:** Loss function depends on entire training data set!

**Thus:** Perform stochastic gradient descent

- Randomly choose one examples \( i \) to compute the loss function
- Update the parameters as in normal gradient descent
- Continue until convergence
MLP: Learning (2)

\[ \ell(D, \hat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\vec{x}_i))^2} \]

Observation: We need to take the derivative of the loss function
But: Loss functions looks complicated
Observation 1: Square-Root is monotone
Observation 2: Loss function depends on entire training data set!
Thus: Perform stochastic gradient descent
  - Randomly choose one examples \( i \) to compute the loss function
  - Update the parameters as in normal gradient descent
  - Continue until convergence
Note: For \( \alpha \to 0 \) it "almost surely" converges
MLP: Learning (3)

New loss function:

\[ \ell(D, \hat{w}) = \frac{1}{2} \left( y_i - \hat{f}(\vec{x}_i) \right)^2 \]

\[ \nabla_{\hat{w}} \ell(D, \hat{w}) = \frac{1}{2} 2(y_i - \hat{f}(\vec{x}_i)) \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \]

Observation:
We need to compute derivative
\[ \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \]

\[ \hat{f}(\vec{x}) = \begin{cases} +1 & \text{if } \sum d_i = 1 \quad w_i \cdot x_i + b \geq 0 \\ 0 & \text{else} \end{cases} \]

Observation:
\( f \) is not continuous in \( 0 \) (it makes a step)
Thus:
Impossible to derive \( \nabla_{\hat{w}} \ell(D, \hat{w}) \) in \( 0 \), because \( f \) is not differentiable in \( 0 \)!
MLP: Learning (3)

New loss function:

\[ \ell(D, \hat{w}) = \frac{1}{2} \left( y_i - \hat{f}(\vec{x}_i) \right)^2 \]

\[ \nabla_{\hat{w}} \ell(D, \hat{w}) = \frac{1}{2} 2(y_i - \hat{f}(\vec{x}_i)) \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \]

Observation: We need to compute derivative \( \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \)
MLP: Learning (3)

New loss function:

\[ \ell(D, \hat{w}) = \frac{1}{2} \left( y_i - \hat{f}(\vec{x}_i) \right)^2 \]

\[ \nabla_{\hat{w}} \ell(D, \hat{w}) = \frac{1}{2} 2(y_i - \hat{f}(\vec{x}_i)) \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \]

Observation: We need to compute derivative \( \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \)

\[ \hat{f}(\vec{x}) = \begin{cases} +1 & \text{if } \sum_{i=1}^{d} w_i \cdot x_i + b \geq 0 \\ 0 & \text{else} \end{cases} \]
MLP: Learning (3)

New loss function:

\[
\ell(\mathcal{D}, \hat{w}) = \frac{1}{2} \left( y_i - \hat{f}(\vec{x}_i) \right)^2
\]

\[
\nabla_{\hat{w}} \ell(\mathcal{D}, \hat{w}) = \frac{1}{2} 2(y_i - \hat{f}(\vec{x}_i)) \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}}
\]

Observation: We need to compute derivative \( \frac{\partial \hat{f}(\vec{x}_i)}{\partial \hat{w}} \)

\[
\hat{f}(\vec{x}) = \begin{cases} 
+1 & \text{if } \sum_{i=1}^{d} w_i \cdot x_i + b \geq 0 \\
0 & \text{else}
\end{cases}
\]

Observation: \( f \) is not continuous in 0 (it makes a step)

Thus: Impossible to derive \( \nabla_{\hat{w}} \ell(\mathcal{D}, w) \) in 0, because \( f \) is not differentiable in 0!
MLP: Activation function

Solution: We need to make $f$ continuous
MLP: Activation function

**Solution:** We need to make $f$ continuous

**Bonus:** This seems to be a little closer to real neurons

**Bonus 2:** We have non-linearity inside the network (more later)
MLP: Activation function

Solution: We need to make \( f \) continuous

Bonus: This seems to be a little closer to real neurons

Bonus 2: We have non-linearity inside the network (more later)

Idea: Use sigmoid activation function

\[
\sigma(z) = \frac{1}{1 + e^{-\beta z}}, \beta \in \mathbb{R} > 0
\]

Note: \( \beta \) controls slope around 0
Sigmoid activation function: Derivative

**Given:** \( \sigma(z) = \frac{1}{1 + e^{-\beta \cdot z}}, \beta \in \mathbb{R}_{>0} \)
Sigmoid activation function: Derivative

**Given:** \( \sigma(z) = \frac{1}{1 + e^{-\beta z}}, \beta \in \mathbb{R}_{>0} \)

**Derivative:**

\[
\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z} \left(1 + e^{-\beta z}\right)^{-1} = (-1) \left(1 + e^{-\beta z}\right)^{-2} (-\beta) e^{-\beta z}
\]
Sigmoid activation function: Derivative

Given: \( \sigma(z) = \frac{1}{1 + e^{-\beta z}}, \beta \in \mathbb{R} > 0 \)

Derivative:

\[
\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z} \left( 1 + e^{-\beta z} \right)^{-1} = (-1) \left( 1 + e^{-\beta z} \right)^{-2} (-\beta) e^{-\beta z}
\]

\[
= \frac{\beta e^{-\beta z}}{(1 + e^{-\beta z})^2} = \beta \frac{e^{-\beta z}}{1 + e^{-\beta z}} \frac{1}{1 + e^{-\beta z}}
\]
Sigmoid activation function: Derivative

**Given:** \( \sigma(z) = \frac{1}{1+e^{-\beta z}}, \beta \in \mathbb{R}_{>0} \)

**Derivative:**

\[
\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z} \left(1 + e^{-\beta z}\right)^{-1} = (-1) \left(1 + e^{-\beta z}\right)^{-2} (-\beta) e^{-\beta z}
\]

\[
= \frac{\beta e^{-\beta z}}{(1 + e^{-\beta z})^2} = \beta \frac{e^{-\beta z}}{1 + e^{-\beta z}} \frac{1}{1 + e^{-\beta z}}
\]

\[
= \beta \frac{e^{-\beta z} + 1 - 1}{1 + e^{-\beta z}} \frac{1}{1 + e^{-\beta z}}
\]
Sigmoid activation function: Derivative

**Given:** \( \sigma(z) = \frac{1}{1 + e^{-\beta z}}, \beta \in \mathbb{R}_{>0} \)

**Derivative:**

\[
\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z} \left( 1 + e^{-\beta z} \right)^{-1} = (-1) \left( 1 + e^{-\beta z} \right)^{-2} (-\beta) e^{-\beta z}
\]

\[
= \frac{\beta e^{-\beta z}}{(1 + e^{-\beta z})^2} = \beta \frac{e^{-\beta z}}{1 + e^{-\beta z}} \frac{1}{1 + e^{-\beta z}}
\]

\[
= \beta \left( \frac{1 + e^{-\beta z}}{1 + e^{-\beta z}} - \frac{1}{1 + e^{-\beta z}} \right) \frac{1}{1 + e^{-\beta z}}
\]
Sigmoid activation function: Derivative

Given: \( \sigma(z) = \frac{1}{1 + e^{-\beta z}}, \beta \in \mathbb{R} > 0 \)

Derivative:

\[
\frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z} \left(1 + e^{-\beta z}\right)^{-1} = (-1) \left(1 + e^{-\beta z}\right)^{-2} (-\beta) e^{-\beta z}
\]

\[
= \frac{\beta e^{-\beta z}}{(1 + e^{-\beta z})^2} = \beta \frac{e^{-\beta z}}{1 + e^{-\beta z}} \frac{1}{1 + e^{-\beta z}}
\]

\[
= \beta \left(\frac{1 + e^{-\beta z}}{1 + e^{-\beta z}} - \frac{1}{1 + e^{-\beta z}}\right) \frac{1}{1 + e^{-\beta z}}
\]

\[
= \beta (1 - \sigma(z)) \sigma(z)
\]
MLP: Activation function (2)

**But:** Binary classification assumes $\mathcal{Y} = \{0, +1\}$
MLP: Activation function (2)

But: Binary classification assumes $\mathcal{Y} = \{0, +1\}$

Thus: Given $L$ layer in total

- **Internally:** We use $f_j^{(l+1)} = \sigma \left( \sum_{i=0}^{M(l)} w_{i,j}^{(l+1)} f_i^{(l)} + b_j^{(l+1)} \right)$

- **Prediction:** Is mapped to 0 or 1:

$$\hat{f}(\vec{x}) = \begin{cases} 
+1 & \text{if } \sigma \left( \sum_{i=0}^{M(L-1)} w_i^{(L)} f_i^{(L-1)} + b^{(L)} \right) \geq 0 \\
0 & \text{else}
\end{cases}$$
MLP: Activation function (2)

**But:** Binary classification assumes $\mathcal{Y} = \{0, +1\}$

**Thus:** Given $L$ layer in total

- **Internally:** We use
  $$f_{j}^{(l+1)} = \sigma \left( \sum_{i=0}^{M^{(l)}} w_{i,j}^{(l+1)} f_{i}^{(l)} + b_{j}^{(l+1)} \right)$$

- **Prediction:** Is mapped to 0 or 1:
  $$\hat{f}(x) = \begin{cases} 
  +1 & \text{if } \sigma \left( \sum_{i=0}^{M^{(L-1)}} w_{i}^{(L)} f_{i}^{(L-1)} + b^{(L)} \right) \geq 0 \\
  0 & \text{else} \end{cases}$$

**Learning with gradient descent:**

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \frac{\partial \ell}{\partial w_{i,j}^{(l)}}$$

$$b_{j}^{(l)} = b_{j}^{(l)} - \alpha \cdot \frac{\partial \ell}{\partial b_{j}^{(l)}}$$
MLP: Notation Recap

**Note:** Too many $l$ and $\ell$’s: Use $E = \ell$ (loss) for easier reading

\[
\begin{align*}
\text{find} & : \quad \frac{\partial E}{\partial w_{i,j}}^{(l)}, \quad \frac{\partial E}{\partial b_j}^{(l)} \\
M^{(l)} & \equiv \# \text{Neurons in layer } l \\
y_j^{(l+1)} & = \sum_{i=0}^{M^{(l)}} w_{i,j}^{(l+1)} f_i^{(l)} + b_j^{(l+1)} \\
f_j^{(l+1)} & = \sigma \left(y_j^{(l+1)}\right) \\
\sigma(z) & = \frac{1}{1 + e^{-\beta \cdot z}}, \beta = 1
\end{align*}
\]
Backpropagation for sigmoid activation / RMSE loss

Gradient step:

\[
\begin{align*}
    w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\
    b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)}
\end{align*}
\]

Recursion:

\[
\begin{align*}
    \delta_j^{(l-1)} &= f_j^{(l-1)} \left(1 - f_j^{(l-1)}\right) \left\{ \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \right\} \\
    \delta_j^{(L)} &= - \left(y_i - f_j^{(L)}\right) f_j^{(L)} \left(1 - f_j^{(L)}\right)
\end{align*}
\]
Backpropagation for sigmoid activation / RMSE loss

**Gradient step:**

\[
\begin{align*}
    w^{(l)}_{i,j} &= w^{(l)}_{i,j} - \alpha \cdot \delta^{(l)}_j f^{(l-1)}_i \\
    b^{(l)}_j &= b^{(l)}_j - \alpha \cdot \delta^{(l)}_j
\end{align*}
\]

**Recursion:**

\[
\delta^{(l-1)}_j = f^{(l-1)}_j \left( 1 - f^{(l-1)}_j \right) \sum_{k=1}^{M^{(l)}} \delta^{(l)}_k w^{(l)}_{j,k}
\]

\[
\delta^{(L)}_j = - \left( y_i - f^{(L)}_j \right) f^{(L)}_j \left( 1 - f^{(L)}_j \right)
\]
Backpropagation for activation $h$ / loss $\ell$

**Gradient step:**

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)}$$

$$b_j^{(l)} = b_j^{(l)} - \alpha \cdot \delta_j^{(l)}$$

**Recursion:**

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_j^{(l)} w_{j,k}^{(l)}$$

$$\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}}$$
Backpropagation: Different notation

**Notation:** We used scalar notation so far

**Fact:** Same results can be derived using matrix-vector notation

→ Notation depends on your preferences and background
Backpropagation: Different notation

**Notation:** We used scalar notation so far

**Fact:** Same results can be derived using matrix-vector notation

→ Notation depends on your preferences and background

**For us:** We want to implement backprop. from scratch, thus scalar notation is closer to our implementation

**But:** Literature usually use matrix-vector notation for compactness
Backpropagation: Different notation

**Notation**: We used scalar notation so far

**Fact**: Same results can be derived using matrix-vector notation

→ Notation depends on your preferences and background

**For us**: We want to implement backprop. from scratch, thus scalar notation is closer to our implementation

**But**: Literature usually use matrix-vector notation for compactness

\[
\delta^{(l-1)} = \left( W^{(l)} \right)^T \delta^{(l)} \odot \frac{\partial h(y^{(l-1)})}{\partial y^{(l-1)}}
\]

\[
\delta^{(L)} = \nabla_{y^{(L)}} \ell(y^{(L)}) \odot \frac{\partial h(y^{(L)})}{\partial y^{(L)}}
\]
Backpropagation: Different notation

**Notation:** We used scalar notation so far

**Fact:** Same results can be derived using matrix-vector notation

→ Notation depends on your preferences and background

**For us:** We want to implement backprop. from scratch, thus scalar notation is closer to our implementation

**But:** Literature usually use matrix-vector notation for compactness

\[
\delta^{(l-1)} = \left( W^{(l)} \right)^T \delta^{(l)} \odot \frac{\partial h(y^{(l-1)})}{\partial y^{(l-1)}}
\]

\[
\delta^{(L)} = \nabla_{y^{(L)}} \ell(y^{(L)}) \odot \frac{\partial h(y^{(L)})}{\partial y^{(L)}}
\]

Hadamard-product / Schur-product: piecewise multiplication

vectorial derivative!
Backpropagation: Some implementation ideas

**Observation:** Backprop. is independent from activation $h$ and loss $\ell$
Backpropagation: Some implementation ideas

**Observation:** Backprop. is independent from activation $h$ and loss $\ell$

**Thus:** Implement neural networks layer-wise:

- Each layer / neuron has activation function
- Each layer / neuron has derivative of activation function
- Each layer has weight matrix (either for input or output)
- Each layer implements delta computation
- Output-layer implements delta computation with loss function
- Layers are either connected to each other and recursively call backprop. or some “control” function performs backprop.
Backpropagation: Some implementation ideas

**Observation:** Backprop. is independent from activation $h$ and loss $\ell$

**Thus:** Implement neural networks layer-wise:

- Each layer / neuron has activation function
- Each layer / neuron has derivative of activation function
- Each layer has weight matrix (either for input or output)
- Each layer implements delta computation
- Output-layer implements delta computation with loss function
- Layers are either connected to each other and recursively call backprop. or some “control” function performs backprop.

**Thus:** Arbitrary network architectures can be realised without changing learning algorithm
Network architectures

**Question:** So what is a good architecture?

- Depends on the problem. Usually, architectures for new problems are published in scientific papers or even as PHD thesis.
- **Non-linear activation:** A network should contain at least one layer with non-linear activation function for better learning.
- **Sparse activation:** To prevent over-fitting, only a few neurons of the network should be active at the same time.
- **Fast convergence:** The loss function / activation function should allow a fast convergence in the first few epochs.
- **Feature extraction:** Combining multiple layers in deeper networks usually allows (higher) level feature extraction.
Network architectures

**Question:** So what is a good architecture?

**Answer:** Depends on the problem. Usually, architectures for new problems are published in scientific papers or even as PHD thesis.
Network architectures

**Question:** So what is a good architecture?
**Answer:** Depends on the problem. Usually, architectures for new problems are published in scientific papers or even as PHD thesis.

**Some general ideas:**

- **Non-linear activation:** A network should contain at least one layer with non-linear activation function for better learning
- **Sparse activation:** To prevent over-fitting, only a few neurons of the network should be active at the same time
- **Fast convergence:** The loss function / activation function should allow a fast convergence in the first few epochs
- **Feature extraction:** Combining multiple layers in deeper networks usually allows (higher) level feature extraction
Backpropagation: Vanishing gradients

**Observation 1:** \( \sigma(z) = \frac{1}{1 + e^{-\beta \cdot z}} \in [0, 1] \)

**Observation 2:** \( \frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) \in [0, 1] \)

**Observation 3:** Errors are multiplied from the next layer
Backpropagation: Vanishing gradients

Observation 1: \( \sigma(z) = \frac{1}{1+e^{-\beta \cdot z}} \in [0, 1] \)
Observation 2: \( \frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) \in [0, 1] \)
Observation 3: Errors are multiplied from the next layer

Thus: The error tends to become very small after a few layers
⇒ The gradient vanishes in each layer more and more
Backpropagation: Vanishing gradients

**Observation 1:** \( \sigma(z) = \frac{1}{1 + e^{-\beta \cdot z}} \in [0, 1] \)

**Observation 2:** \( \frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) \in [0, 1] \)

**Observation 3:** Errors are multiplied from the next layer

**Thus:** The error tends to become very small after a few layers

\( \Rightarrow \) The gradient vanishes in each layer more and more

**So far:** No fundamental solution found, but a few suggestions

- Change activation function
- Exploit different optimization methods
- Use more data / carefully adjust stepsizes
- Reduce number of parameters / depth of network
New activation function: ReLu

Rectified Linear (ReLu):
New activation function: ReLu

Rectified Linear (ReLu):

\[
h(z) = \begin{cases} 
  z & \text{if } z \geq 0 \\
  0 & \text{else}
\end{cases} = \max(0, z)
\]

\[
\frac{\partial h(z)}{\partial z} = \begin{cases} 
  1 & \text{if } z \geq 0 \\
  0 & \text{else}
\end{cases}
\]

Note: ReLu is not differentiable in \(z = 0\)!

But: Usually that is not a problem

Practical: \(z = 0\) is pretty rare, just use 0 there. It works well

Mathematical: There exists a subgradient of \(h(z)\) at 0

DeepLearning on FPGAs
New activation function: ReLu

Rectified Linear (ReLu):

$$h(z) = \begin{cases} 
  z & \text{if } z \geq 0 \\
  0 & \text{else} 
\end{cases} = \max(0, z)$$

$$\frac{\partial h(z)}{\partial z} = \begin{cases} 
  1 & \text{if } z \geq 0 \\
  0 & \text{else} 
\end{cases}$$

**Note:** ReLu is not differentiable in $z = 0$!
New activation function: ReLu

Rectified Linear (ReLu):

\[ h(z) = \begin{cases} 
  z & \text{if } z \geq 0 \\
  0 & \text{else} 
\end{cases} = \max(0, z) \]

\[ \frac{\partial h(z)}{\partial z} = \begin{cases} 
  1 & \text{if } z \geq 0 \\
  0 & \text{else} 
\end{cases} \]

**Note:** ReLu is not differentiable in \( z = 0 \)!

**But:** Usually that is not a problem

- **Practical:** \( z = 0 \) is pretty rare, just use 0 there. It works well
- **Mathematical:** There exists a subgradient of \( h(z) \) at 0
ReLu(2)

Subgradients: A gradient shows the direct of the steepest descent
⇒ If a function is not differentiable, it has no steepest descent
⇒ There might be multiple (equally) “steepest descents”
ReLu(2)

Subgradients: A gradient shows the direct of the steepest descent
⇒ If a function is not differentiable, it has no steepest descent
⇒ There might be multiple (equally) “steepest descents”

For ReLu: We can choose \[ \frac{\partial h(z)}{\partial z} \bigg|_{z=0} \] from \([0, 1]\)

Big Note: Using a subgradient does not guarantee that our loss function decreases! We might change weights to the worse!
ReLu(2)

**Subgradients:** A gradient shows the direct of the steepest descent
⇒ If a function is not differentiable, it has no steepest descent
⇒ There might be multiple (equally) “steepest descents”

**For ReLu:** We can choose \( \frac{\partial h(z)}{\partial z} \mid_{z=0} \) from \([0, 1]\)

**Big Note:** Using a subgradient does not guarantee that our loss function decreases! We might change weights to the worse!

**Nice properties of ReLu:**
- Super-easy forward, backward and derivative computation
- Either activates or deactivates a neuron (sparsity)
- Less problems with gradient vanishing, since error is multiplied by 1 or 0
- Still gives network non-linear activation
Improve convergence for GD: Simple improvements

**Gradient descent:**

\[ \hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old}) \]
Improve convergence for GD: Simple improvements

\textbf{Gradient descent:}

\[ \hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old}) \]

\textbf{Momentum:} Keep the momentum from previous updates

\[ \Delta \hat{\theta}^{old} = \alpha_1 \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old}) + \alpha_2 \Delta \hat{\theta}^{old} \]

\[ \hat{\theta}^{new} = \hat{\theta}^{old} - \Delta \hat{\theta}^{old} \]
Improve convergence for GD: Simple improvements

Gradient descent:

$$\hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old})$$

Momentum: Keep the momentum from previous updates

$$\Delta \hat{\theta}^{old} = \alpha_1 \cdot \nabla_{\theta} \ell(D, \hat{\theta}^{old}) + \alpha_2 \Delta \hat{\theta}^{old}$$

$$\hat{\theta}^{new} = \hat{\theta}^{old} - \Delta \hat{\theta}^{old}$$

(Mini-)Batch: Compute derivatives for multiple examples and average direction (allows parallel computation of gradient)

$$\hat{\theta}^{new} = \hat{\theta}^{old} - \alpha \cdot \frac{1}{K} \sum_{i=0}^{K} \nabla_{\theta} \ell(\vec{x}_i, \hat{\theta}^{old})$$

Note: For Mini-Batch approaches the convergence is not guaranteed theoretically
Improve convergence: Stepsize

What about the stepsize?

- If it's too small, you will learn slow (→ more data required)
- If it's too big, you might miss the optimum (→ bad results)

Note: We can always reuse our data (multiple passes over dataset)

But: Step size is problem specific as always!

Practical suggestion: Simple heuristic
- Try out different stepsizes on small subsample of data
- Pick that one that most reduces the loss
- Use it for on the full dataset

Sidenote: Changing the step size while training also possible
Improve convergence: Stepsize

What about the stepsize?

- If its too small, you will learn slow (∞ more data required)
- If its too big, you might miss the optimum (∞ bad results)

Thus usually: Small $\alpha = 0.001 - 0.1$ with a lot of data

Note: We can always reuse our data (multiple passes over dataset)

But: Stepsize is problem specific as always!
Improve convergence: Step size

What about the stepsize?

- If it's too small, you will learn slow (→ more data required)
- If it's too big, you might miss the optimum (→ bad results)

Thus usually: Small $\alpha = 0.001 - 0.1$ with a lot of data

Note: We can always reuse our data (multiple passes over dataset)

But: Stepsize is problem specific as always!

Practical suggestion: Simple heuristic

- Try out different stepsizes on small subsample of data
- Pick that one that most reduces the loss
- Use it for on the full dataset

Sidenote: Changing the stepsize while training also possible
Improve convergence: Loss functions

Recap: $\delta_j^{(L)}$ should be relatively large for faster learning:

$$
\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}
$$
Improve convergence: Loss functions

**Recap:** \( \delta_j^{(L)} \) should be relatively large for faster learning:

\[
\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}.
\]

- Squared error:
  \[
  \ell(D, \hat{\theta}) = \frac{1}{2} (y - \hat{y})^2 \Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -(y - \hat{y})
  \]
  \[
  \delta_j^{(L)} = -(y - \hat{y}) \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}
  \]
  tends to be small if \( h \) is sigmoid

- Cross-entropy:
  \[
  \ell(D, \hat{\theta}) = -\left( y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}) \right)
  \]
  \[
  \frac{\partial \ell}{\partial \hat{y}} = -y + 1 - y = \hat{y} - y \]
  \[
  \delta_j^{(L)} = \hat{y} - y \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}
  \]
  tends to be small if \( h \) is sigmoid
Improve convergence: Loss functions

Recap: $\delta_j^{(L)}$ should be relatively large for faster learning:

$$\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}$$

Squared error: $\ell(D, \hat{\theta}) = \frac{1}{2} (y - \hat{y})^2 \Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -(y - \hat{y})$

$\Rightarrow \delta_j^{(L)} = -(y - \hat{y}) \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}$ is still small if sigmoid is used
Improve convergence: Loss functions

Recap: $\delta_j^{(L)}$ should be relatively large for faster learning:

$$
\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}.
$$

Squared error: $\ell(D, \hat{\theta}) = \frac{1}{2} (y - \hat{y})^2 \Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -(y - \hat{y})$

$\Rightarrow \delta_j^{(L)} = -(y - \hat{y}) \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}$ is still small if sigmoid is used

Cross-entropy: $\ell(D, \hat{\theta}) = -(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$

$\Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{(1 - \hat{y})\hat{y}}$

Tends to be small if $h$ is sigmoid
Improve convergence: Loss functions

Recap: $\delta_j^{(L)}$ should be relatively large for faster learning:

$$
\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}.
$$

Squared error: $\ell(D, \hat{\theta}) = \frac{1}{2} (y - \hat{y})^2 \Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -(y - \hat{y})$

$\rightarrow \delta_j^{(L)} = -(y - \hat{y}) \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}$ is still small if sigmoid is used

Cross-entropy: $\ell(D, \hat{\theta}) = -(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$

$$
\Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{(1 - \hat{y})\hat{y}}
$$

tends to be small if $h$ is sigmoid

derivative of sigmoid function
Improve convergence: Loss functions

Recap: $\delta_j^{(L)}$ should be relatively large for faster learning:

$$
\delta_j^{(L)} = \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}} = \frac{\partial \ell(\hat{y})}{\partial \hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}
$$

Squared error: $\ell(D, \hat{\theta}) = \frac{1}{2} (y - \hat{y})^2 \Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -(y - \hat{y})$

$\rightarrow \delta_j^{(L)} = -(y - \hat{y}) \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}}$ is still small if sigmoid is used

Cross-entropy: $\ell(D, \hat{\theta}) = -(y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y}))$

$\Rightarrow \frac{\partial \ell}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{(1 - \hat{y})\hat{y}}$

$\rightarrow \delta_j^{(L)} = \frac{\hat{y} - y}{(1 - \hat{y})\hat{y}} \cdot \frac{\partial h(\hat{y})}{\partial \hat{y}} = \hat{y} - y$ cancels small sigmoid values
Improve Convergence: Start solution

Where do we start?
In SGD: Start with some $\theta$. SGD will walk us the right direction
Important: For NN (specifically for MSE + sigmoid activation) we need “sane” initialization:

$$\delta_j^{(L)} = - (y_i - f_j^{(L)}) f_j^{(L)} (1 - f_j^{(L)})$$

$$\Rightarrow \delta_j^{(L)} = 0, \text{ if } f_j^{(L)} = 0 \text{ or } f_j^{(L)} = 1$$

Therefore: Init weights randomly with gaussian distribution

$$w_{i,j}^{(l)} \sim \mathcal{N}(0, \varepsilon) \text{ with } \varepsilon = 0.001 - 0.1$$

Bonus: Negative weights are also present
Summary

Important concepts:
- For parameter optimization we define a loss function.
- For parameter optimization we use gradient descent.
- Neurons have activation functions to ensure non-linearity and differentiability.
- Backpropagation is an algorithm to compute the gradient.
- Non-linear and sparse networks are usually better.
- Various techniques can be used to improve convergence speed.
Homework

Homework until next meeting

- Implement the following network to solve the XOR problem

\[ x_1 \]
\[ x_2 \]

- Implement backpropagation for this network
  - Try a simple solution first: Hardcode one activation / one loss function with fixed access to data structures
  - If you feel comfortable, add new activation / loss functions

Tip 1: Verify that the proposed network uses 9 parameters
Tip 2: Start with \( \alpha = 1.0 \) and 10000 training examples

Note: We will later use C, so please use C or a C-like language

Question: Can you reduce the number of examples necessary?