DeepLearning on FPGAs

Artificial Neural Networks: Backpropagation and more

Sebastian Buschjäger

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

November 10, 2016

Recap: Homework

Question: So whats your accuracy?
Question: What about speed?

Recap: Homework

Question: So whats your accuracy?
Question: What about speed?

Some remark about notation: In the previous slides I used θ 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. θ 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 ($\rightarrow$ perceptron)
  - Specify optimization procedure ($\rightarrow$ gradient descent)
  - Specify a cost / loss function ($\rightarrow$ 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 ($\rightarrow$ perceptron)
  - Specify optimization procedure ($\rightarrow$ gradient descent)
  - Specify a cost / loss function ($\rightarrow$ RMSE or cross-entropy)

Perceptron: A linear classifier $f\colon \mathbb{R}^d \rightarrow \{0, 1\}$ with

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

Recap: Data Mining (2)

Optimization procedure: Gradient descent

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

Recap: Data Mining (2)

Optimization procedure: Gradient descent

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

Loss function: RMSE or cross-entropy

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

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

Recap: Data Mining (2)

Optimization procedure: Gradient descent

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

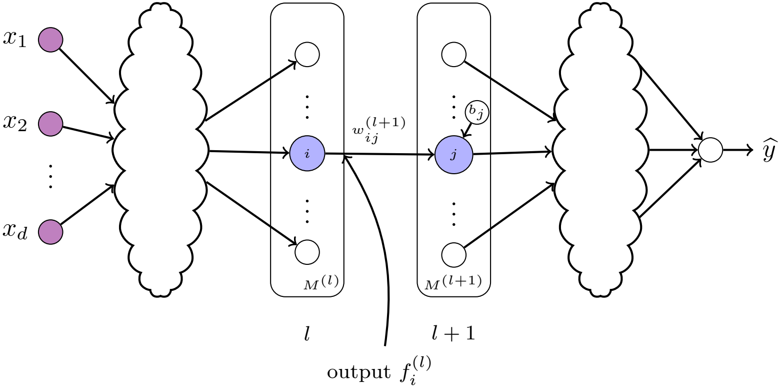
Loss function: RMSE or cross-entropy

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

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

So far: Training of single perceptron
Now: Training of multi-layer perceptron (MLP)

MLP: Some Notation (1)



$\mathbf{w_{i,j}^{(l+1)}} \widehat{=}$ Weight from neuron i in layer l to neuron j in layer $l+1$

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

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:

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

Loss function:

$$\ell(\mathcal{D}, \widehat{w}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \Big(y_i - \widehat{f}(\vec{x}_i)\Big)^2}$$

MLP: Learning (2)

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

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

MLP: Learning (2)

$$\ell(\mathcal{D}, \widehat{w}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\Big(y_i - \widehat{f}(\vec{x}_i)\Big)^2}$$

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)

$$\ell(\mathcal{D}, \widehat{w}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\Big(y_i - \widehat{f}(\vec{x}_i)\Big)^2}$$

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)

$$\ell(\mathcal{D}, \widehat{w}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\Big(y_i - \widehat{f}(\vec{x}_i)\Big)^2}$$

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:

$$
\begin{aligned}
\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}\left(y_i - \widehat{f}(\vec{x}_i)\right)^2 \\
\nabla_{\widehat{w}}\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}2(y_i - \widehat{f}(\vec{x}_i))\frac{\partial \widehat{f}(\vec{x}_i)}{\partial \widehat{w}}
\end{aligned}
$$

MLP: Learning (3)

New loss function:

$$
\begin{aligned}
\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}\left(y_i - \widehat{f}(\vec{x}_i)\right)^2 \\
\nabla_{\widehat{w}}\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}2(y_i - \widehat{f}(\vec{x}_i))\frac{\partial \widehat{f}(\vec{x}_i)}{\partial \widehat{w}}
\end{aligned}
$$

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

MLP: Learning (3)

New loss function:

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

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

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

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

MLP: Learning (3)

New loss function:

$$\begin{aligned}\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}\Big(y_i - \widehat{f}(\vec{x}_i)\Big)^2 \\ \nabla_{\widehat{w}}\ell(\mathcal{D}, \widehat{w}) &= \frac{1}{2}2(y_i - \widehat{f}(\vec{x}_i))\frac{\partial \widehat{f}(\vec{x}_i)}{\partial \widehat{w}}\end{aligned}$$

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

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

Observation: f is not continuous in 0 (it makes a step)
Thus: Impossible to derive $\nabla_{\widehat{w}}\ell(\mathcal{D}, w)$ 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)

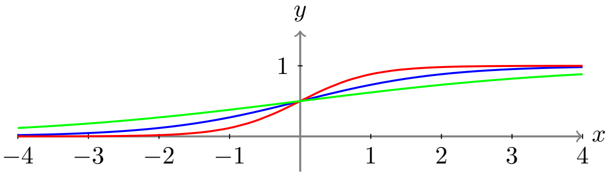
MLP: Activation function

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



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

Note: β 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: $\sigma(z) = \frac{1}{1+e^{-\beta \cdot z}}, \beta \in \mathbb{R}_{>0}$
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: $\sigma(z) = \frac{1}{1+e^{-\beta \cdot z}}, \beta \in \mathbb{R}_{>0}$
Derivative:

$$\begin{aligned} \frac{\partial \sigma(z)}{\partial z} &= \frac{\partial}{\partial z}\left(1+e^{-\beta z}\right)^{-1} = (-1)\left(1+e^{-\beta z}\right)^{-2}(-\beta)e^{-\beta z} \\ &= \frac{\beta e^{-\beta z}}{\left(1+e^{-\beta z}\right)^2} = \beta \frac{e^{-\beta z}}{1+e^{-\beta z}} \frac{1}{1+e^{-\beta z}} \end{aligned}$$

Sigmoid activation function: Derivative

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

$$\begin{aligned}
\frac{\partial\sigma(z)}{\partial z} &= \frac{\partial}{\partial z}\left(1+e^{-\beta z}\right)^{-1} = (-1)\left(1+e^{-\beta z}\right)^{-2}(-\beta)e^{-\beta z} \\
&= \frac{\beta e^{-\beta z}}{(1+e^{-\beta z})^2} \qquad = \beta\frac{e^{-\beta z}}{1+e^{-\beta z}}\frac{1}{1+e^{-\beta z}} \\
&= \beta\frac{e^{-\beta z}+1-1}{1+e^{-\beta z}}\frac{1}{1+e^{-\beta z}}
\end{aligned}$$

Sigmoid activation function: Derivative

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

$$\begin{aligned}
\frac{\partial \sigma(z)}{\partial z} &= \frac{\partial}{\partial z}\left(1+e^{-\beta z}\right)^{-1} = (-1)\left(1+e^{-\beta z}\right)^{-2}(-\beta)e^{-\beta z} \\
&= \frac{\beta e^{-\beta z}}{\left(1+e^{-\beta z}\right)^2} \qquad = \beta \frac{e^{-\beta z}}{1+e^{-\beta z}} \frac{1}{1+e^{-\beta z}} \\
&= \beta \frac{e^{-\beta z}+1-1}{1+e^{-\beta z}} \frac{1}{1+e^{-\beta z}} \\
&= \beta \left(\frac{1+e^{-\beta z}}{1+e^{-\beta z}} - \frac{1}{1+e^{-\beta z}}\right) \frac{1}{1+e^{-\beta z}}
\end{aligned}$$

Sigmoid activation function: Derivative

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

$$\begin{aligned}
\frac{\partial \sigma(z)}{\partial z} &= \frac{\partial}{\partial z}\left(1+e^{-\beta z}\right)^{-1} = (-1)\left(1+e^{-\beta z}\right)^{-2}(-\beta)e^{-\beta z} \\
&= \frac{\beta e^{-\beta z}}{(1+e^{-\beta z})^2} \qquad = \beta \frac{e^{-\beta z}}{1+e^{-\beta z}} \frac{1}{1+e^{-\beta z}} \\
&= \beta \frac{e^{-\beta z}+1-1}{1+e^{-\beta z}} \frac{1}{1+e^{-\beta z}} \\
&= \beta \left(\frac{1+e^{-\beta z}}{1+e^{-\beta z}} - \frac{1}{1+e^{-\beta z}} \right) \frac{1}{1+e^{-\beta z}} \\
&= \beta(1-\sigma(z))\sigma(z)
\end{aligned}$$

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:

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

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:

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

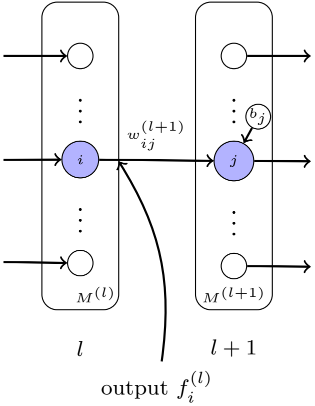
Learning with gradient descent:

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

$$b_j^{(l)} = b_j^{(l)} - \alpha \cdot \frac{\partial \ell}{\partial b_j^{(l)}}$$

MLP: Notation Recap

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



$$\textbf{find} : \frac{\partial E}{\partial w_{i,j}^{(l)}}, \frac{\partial E}{\partial b_j^{(l)}}$$

$$M^{(l)} \;\hat{=}\; \#\text{Neurons in layer } l$$

$$y_j^{(l+1)} = \sum_{i=0}^{M^{(l)}} w_{i,j}^{(l+1)} f_i^{(l)} + b_j^{(l+1)}$$

$$f_j^{(l+1)} = \sigma\left(y_j^{(l+1)}\right)$$

$$\sigma(z) = \frac{1}{1+e^{-\beta \cdot z}}, \beta = 1$$

. . .

. . .

. . .

Backpropagation for sigmoid activation / RMSE loss

Gradient step:

$$
\begin{aligned}
w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\
b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)}
\end{aligned}
$$

Recursion:

$$
\begin{aligned}
\delta_j^{(l-1)} &= f_j^{(l-1)} \left(1 - f_j^{(l-1)}\right) \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \\
\delta_j^{(L)} &= -\left(y_i - f_j^{(L)}\right) f_j^{(L)} \left(1 - f_j^{(L)}\right)
\end{aligned}
$$

Backpropagation for sigmoid activation / RMSE loss

Gradient step:

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

derivative of activation function

Recursion:

$$\delta_j^{(l-1)} = f_j^{(l-1)}\left(1 - f_j^{(l-1)}\right) \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)}$$
$$\delta_j^{(L)} = -\left(y_i - f_j^{(L)}\right) f_j^{(L)}\left(1 - f_j^{(L)}\right)$$

derivative of loss function

Backpropagation for activation h / loss ℓ

Gradient step:

$$
\begin{aligned}
w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} f_i^{(l-1)} \\
b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)}
\end{aligned}
$$

Recursion:

$$
\begin{aligned}
\delta_j^{(l-1)} &= \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)} \\
\delta_j^{(L)} &= \frac{\partial \ell(y_i^{(L)})}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}}
\end{aligned}
$$

Backpropagation: Different notation

Notation: We used scalar notation so far

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

$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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

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

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

vectorial derivative!

Hadamard-product / Schur-product: piecewise multiplication

Backpropagation: Some implementation ideas

Observation: Backprop. is independent from activation h and loss ℓ

Backpropagation: Some implementation ideas

Observation: Backprop. is independent from activation h and loss ℓ
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 ℓ
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?

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
$\Rightarrow$ 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

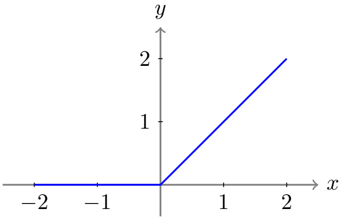
Thus: The error tends to become very small after a few layers
$\Rightarrow$ The gradient vanishes in each layer more and more

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

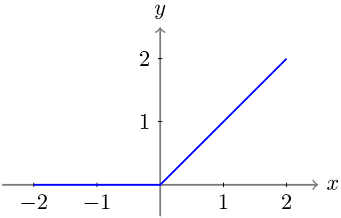
New activation function: ReLu

Rectified Linear (ReLu):



New activation function: ReLu

Rectified Linear (ReLu):

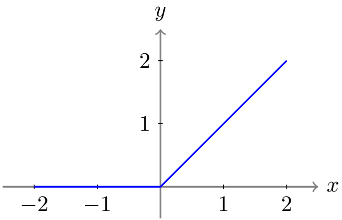


$$h(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases} = max(0, z)$$

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

New activation function: ReLu

Rectified Linear (ReLu):



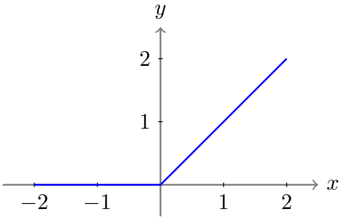
$$h(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases} = max(0, z)$$

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

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
$\Rightarrow$ If a function is not differentiable, it has no steepest descent
$\Rightarrow$ There might be multiple (equally) "steepest descents"

ReLu(2)

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

For ReLu: We can choose $\frac{\partial h(z)}{\partial z}\Big|_{z=0}$ 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
$\Rightarrow$ If a function is not differentiable, it has no steepest descent
$\Rightarrow$ There might be multiple (equally) "steepest descents"

For ReLu: We can choose $\frac{\partial h(z)}{\partial z}\Big|_{z=0}$ 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:

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

Improve convergence for GD: Simple improvements

Gradient descent:

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

Momentum: Keep the momentum from previous updates

$$\begin{aligned} \Delta\widehat{\theta}^{old} &= \alpha_1 \cdot \nabla_\theta \ell(\mathcal{D}, \widehat{\theta}^{old}) + \alpha_2 \Delta\widehat{\theta}^{old} \\ \widehat{\theta}^{new} &= \widehat{\theta}^{old} - \Delta\widehat{\theta}^{old} \end{aligned}$$

Improve convergence for GD: Simple improvements

Gradient descent:

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

Momentum: Keep the momentum from previous updates

$$\begin{aligned}\Delta\widehat{\theta}^{old} &= \alpha_1 \cdot \nabla_\theta \ell(\mathcal{D}, \widehat{\theta}^{old}) + \alpha_2 \Delta\widehat{\theta}^{old} \\ \widehat{\theta}^{new} &= \widehat{\theta}^{old} - \Delta\widehat{\theta}^{old}\end{aligned}$$

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

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

Note: For Mini-Batch approaches the convergence is not guranteed theoretically

Improve convergence: Stepsize

What about the stepsize?

- If its to small, you will learn slow ($\rightarrow$ more data required)
- If its to big, you might miss the optimum ($\rightarrow$ bad results)

Improve convergence: Stepsize

What about the stepsize?

- If its to small, you will learn slow ($\rightarrow$ more data required)
- If its to big, you might miss the optimum ($\rightarrow$ 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: Stepsize

What about the stepsize?

- If its to small, you will learn slow ($\rightarrow$ more data required)
- If its to big, you might miss the optimum ($\rightarrow$ 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:

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

Improve convergence: Loss functions

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

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

tends to be small if h is sigmoid

Improve convergence: Loss functions

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

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

tends to be small if h is sigmoid

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

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

Improve convergence: Loss functions

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

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

tends to be small if h is sigmoid

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

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

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

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

Improve convergence: Loss functions

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

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

tends to be small if h is sigmoid

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

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

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

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

derivative of sigmoid function

Improve convergence: Loss functions

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

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

tends to be small if h is sigmoid

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

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

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

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

derivative of sigmoid function

$\rightarrow \delta_j^{(L)} = \frac{\widehat{y}-y}{(1-\widehat{y})\widehat{y}} \cdot \frac{\partial h(\widehat{y}))}{\partial \widehat{y}} = \widehat{y} - y$ cancels small sigmoid values

Improve Convergence: Start solution

Where do we start?

In SGD: Start with some θ. SGD will walk us the right direction

Important: For NN (specifically for MSE + sigmoid activation) we need "sane" initialization:

$$\begin{aligned} \delta_j^{(L)} &= -\left(y_i - f_j^{(L)}\right) f_j^{(L)} \left(1 - f_j^{(L)}\right) \\ \Rightarrow \delta_j^{(L)} &= 0, \text{ if } f_j^{(L)} = 0 \text{ or } f_j^{(L)} = 1 \end{aligned}$$

Therefore: Init weights randomly with gaussian distribution

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

Bonus: Negative weights are also present

Summary

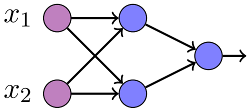
Important concepts:

- **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



- 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?