Regularization

K. Breininger, F. Denzinger, F. Thamm, Z. Yang, N. Maul, F. Meister, C. Liu, S. Jaganathan, L. Folle,
M. Vornehm, A. Popp, B. Geissler, S. Mehltretter, N. Patel, V. Bacher, K. Fischer
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
December 22, 2020

Tasks in this exercise

1. Optimization Constraints: Augmenting the loss function
2. Dropout **Layer**
3. Batch Normalization **Layer**
4. LeNet: Put everything together **(optional)**
5. RNN layer: Elman Unit
6. LSTM layer: Backpropagation at its best! **(optional)**

Optimization Constraints: Loss function augmentation

General outline

- Constraints change the total loss ...
- ... and have influence on the weight update of the respective layer!

General outline

- Constraints change the total loss ...
- ... and have influence on the weight update of the respective layer!

- Implement constraints as separate classes

→ **Independent** of loss function

General outline

- Constraints change the total loss ...
- ... and have influence on the weight update of the respective layer!

- Implement constraints as separate classes

➔ **Independent** of loss function

- Constraints **only need** current weights

➔ Add constraint objects in the optimizer

General outline

- Constraints change the total loss ...
- ... and have influence on the weight update of the respective layer!

- Implement constraints as separate classes

➔ **Independent** of loss function

- Constraints **only need** current weights

➔ Add constraint objects in the optimizer

- Since constraints generate part of the loss:

➔ Change Neural Network container class (and associated classes) to "channel" and gather **regularization loss** for **all layers**

Workflow

- Forward pass

➔ Calculate norm of weights in each trainable layer and gather as regularization loss

➔ Add regularization loss to the final loss

Workflow

- Forward pass

→ Calculate norm of weights in each trainable layer and gather as regularization loss

→ Add regularization loss to the final loss

- Backward pass

→ In each trainable layer, include **the gradient of norm** when calculating update

L_2 regularization

- Forward pass:

$$\tilde{L}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

- Backward pass:

$$\mathbf{w}^{(k+1)} = \underbrace{\left(1 - \eta\lambda\right)\mathbf{w}^{(k)}}_{\text{Shrinkage}} - \eta \frac{\partial L}{\partial \mathbf{w}^{(k)}}$$

L_2 regularization

- Forward pass:

$$\tilde{L}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

- Backward pass:

$$\mathbf{w}^{(k+1)} = \underbrace{\left(1 - \eta\lambda\right)\mathbf{w}^{(k)}}_{\text{Shrinkage}} - \eta \frac{\partial L}{\partial \mathbf{w}^{(k)}}$$

- In the Forward pass the L2 norm gets squared, which eliminates the square root inside and increases the numerical stability as the gradient is easier to compute.

L_2 regularization

- Forward pass:

$$\tilde{L}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

- Backward pass:

$$\mathbf{w}^{(k+1)} = \underbrace{\left(1 - \eta\lambda\right)\mathbf{w}^{(k)}}_{\text{Shrinkage}} - \eta \frac{\partial L}{\partial \mathbf{w}^{(k)}}$$

- In the Forward pass the L2 norm gets squared, which eliminates the square root inside and increases the numerical stability as the gradient is easier to compute.
- Notice for matrices we compute here the Frobenius norm, not the Spectral norm.

L_2 regularization

- Forward pass:

$$\tilde{L}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

- Backward pass:

$$\mathbf{w}^{(k+1)} = \underbrace{\left(1 - \eta\lambda\right)\mathbf{w}^{(k)}}_{\text{Shrinkage}} - \eta \frac{\partial L}{\partial \mathbf{w}^{(k)}}$$

- In the Forward pass the L2 norm gets squared, which eliminates the square root inside and increases the numerical stability as the gradient is easier to compute.
- Notice for matrices we compute here the Frobenius norm, not the Spectral norm.
- The influence of constraints is controlled via λ. Because `lambda` is a python keyword, you want to use e.g. `alpha` instead.

$\mathbf{L_1}$ regularization

- Forward pass:

$$\tilde{L}(\mathbf{w}) = L(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

- Backward pass:

$$\mathbf{w}^{(k+1)} = \underbrace{\mathbf{w}^{(k)} - \eta\lambda \text{sign}\left(\mathbf{w}^{(k)}\right)}_{\text{Other shrinkage}} - \eta \frac{\partial L}{\partial \mathbf{w}^{(k)}}$$

Dropout

Method

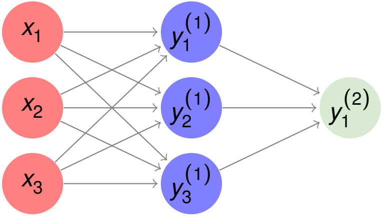

Figure: Dropout

- Implement this as a **fixed-function layer**

Method

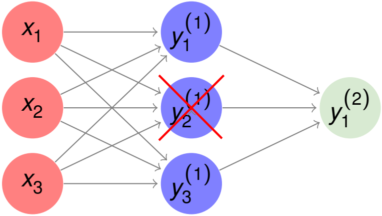

Figure: Dropout

- Implement this as a **fixed-function layer**
- Randomly set **activations** $\mapsto 0$ with probability $1 - p$

Method

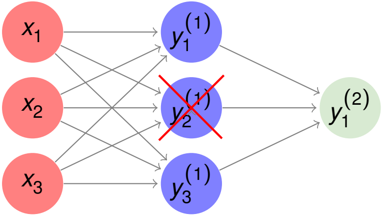

Figure: Dropout

- Implement this as a **fixed-function layer**
- Randomly set **activations** $\mapsto 0$ with probability $1 - p$
- **Test-time**: multiply activations with p

Inverted Dropout

- Can we get rid of the dropout layer at test-time?

Inverted Dropout

- Can we get rid of the dropout layer at test-time?

➔ Change the behavior during training

- Multiply activations in forward-pass **only during training** by $\frac{1}{p}$
- Note: the backward pass has to be adapted as well!

Batch normalization

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ from **batch**

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ from **batch**

$$\hat{\mathbf{Y}} = \gamma \tilde{\mathbf{X}} + \boldsymbol{\beta}$$

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ from **batch**

$$\hat{\mathbf{Y}} = \gamma\tilde{\mathbf{X}} + \beta$$

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ have the **same dimension** as the **input vectors**

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ from **batch**

$$\hat{\mathbf{Y}} = \gamma \tilde{\mathbf{X}} + \boldsymbol{\beta}$$

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ have the **same dimension** as the **input vectors**
- $\boldsymbol{\beta}$, γ and $\boldsymbol{\mu}_B$, $\boldsymbol{\sigma}_B$ have same **dimension** to be able to preserve **identity**

Forward pass

➔ Normalization as a new layer with 2 parameters, γ and β

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ from **batch**

$$\hat{\mathbf{Y}} = \gamma \tilde{\mathbf{X}} + \beta$$

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ have the **same dimension** as the **input vectors**
- $\boldsymbol{\beta}$, γ and $\boldsymbol{\mu}_B$, $\boldsymbol{\sigma}_B$ have same **dimension** to be able to preserve **identity**
- Notice that $\boldsymbol{\beta}$ is a **bias**

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of the **training set**

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of the **training set**
- It's **expensive** to calculate the true training set mean and variance

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of the **training set**
- It's **expensive** to calculate the true training set mean and variance
- Therefore a **moving average** is common:

$$\tilde{\boldsymbol{\mu}}^{(k)} \approx \alpha \tilde{\boldsymbol{\mu}}^{(k-1)} + (1-\alpha)\boldsymbol{\mu}_B^{(k)}$$
$$\tilde{\boldsymbol{\sigma}}^{(k)} \approx \alpha \tilde{\boldsymbol{\sigma}}^{(k-1)} + (1-\alpha)\boldsymbol{\sigma}_B^{(k)}$$

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of the **training set**
- It's **expensive** to calculate the true training set mean and variance
- Therefore a **moving average** is common:

$$\tilde{\boldsymbol{\mu}}^{(k)} \approx \alpha \tilde{\boldsymbol{\mu}}^{(k-1)} + (1-\alpha)\boldsymbol{\mu}_B^{(k)}$$
$$\tilde{\boldsymbol{\sigma}}^{(k)} \approx \alpha \tilde{\boldsymbol{\sigma}}^{(k-1)} + (1-\alpha)\boldsymbol{\sigma}_B^{(k)}$$

- Moving average **decay** α (e.g. 0.8)

Backward pass

- Gradient **with respect to weights** is simply:

$$\frac{\partial L}{\partial \gamma} = \sum_{b=1}^{B} \frac{\partial L}{\partial \hat{\mathbf{Y}}_b} \tilde{\mathbf{X}}_b = \sum_{b=1}^{B} \mathbf{E}_b \tilde{\mathbf{X}}_b$$

- For the **bias** likewise we have:

$$\frac{\partial L}{\partial \beta} = \sum_{b=1}^{B} \frac{\partial L}{\partial \hat{\mathbf{Y}}_b} = \sum_{b=1}^{B} \mathbf{E}_b$$

Backward pass

The **gradient with respect to the input** is more complicated, but here it is:

$$\frac{\partial L}{\partial \tilde{\mathbf{X}}} = \frac{\partial L}{\partial \hat{\mathbf{Y}}} \odot \gamma$$

$$\frac{\partial L}{\partial \boldsymbol{\sigma}_B^2} = \sum_{b=1}^{B} \frac{\partial L}{\partial \tilde{\mathbf{X}}_b} \odot (\mathbf{X}_b - \boldsymbol{\mu}_B) \odot \frac{-1}{2} \left(\boldsymbol{\sigma}_B^2 + \epsilon\right)^{\frac{-3}{2}}$$

$$\frac{\partial L}{\partial \boldsymbol{\mu}_B} = \left(\sum_{b=1}^{B} \frac{\partial L}{\partial \tilde{\mathbf{X}}_b} \odot \frac{-1}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}\right) + \underbrace{\frac{\partial L}{\partial \boldsymbol{\sigma}_B^2} \odot \frac{\sum_{b=1}^{B} -2(\mathbf{X}_b - \boldsymbol{\mu}_B)}{B}}_{0}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \tilde{\mathbf{X}}} \odot \frac{1}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}} + \frac{\partial L}{\partial \boldsymbol{\sigma}_B^2} \odot \frac{2(\mathbf{X} - \boldsymbol{\mu}_B)}{B} + \frac{\partial L}{\partial \boldsymbol{\mu}_B} \odot \frac{1}{B}$$

Backward pass

- $\odot$ denotes an element-wise multiplication. Always check the dimensionality of your matrices!

Backward pass

- $\odot$ denotes an element-wise multiplication. Always check the dimensionality of your matrices!
- To make life easier, we will provide the code for the computation of the gradient with respect to the input:

Backward pass

- $\odot$ denotes an element-wise multiplication. Always check the dimensionality of your matrices!
- To make life easier, we will provide the code for the computation of the gradient with respect to the input:
- `compute_bn_gradients`

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**
- A scalar μ , σ is calculated for the *H* **channels**

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**
- A scalar μ , σ is calculated for the *H* **channels**
- Implementation can be reused, by observing
  - ➔ that **spatial dimensions** *M*, *N* can be treated **like the batch dimension** *B*

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**
- A scalar μ , σ is calculated for the H **channels**
- Implementation can be reused, by observing
  - ➔ that **spatial dimensions** M, N can be treated **like the batch dimension** B
  - ➔ we can **reshape** the $B \times H \times M \times N$ tensor to $B \times H \times M \cdot N$
  - ➔ because of our format we have to **transpose** from $B \times H \times M \cdot N$ to $B \times M \cdot N \times H$
  - ➔ and afterwards **reshape again** to have a $B \cdot M \cdot N \times H$ tensor

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**
- A scalar μ , σ is calculated for the H **channels**
- Implementation can be reused, by observing
  - ➔ that **spatial dimensions** M, N can be treated **like the batch dimension** B
  - ➔ we can **reshape** the $B \times H \times M \times N$ tensor to $B \times H \times M \cdot N$
  - ➔ because of our format we have to **transpose** from $B \times H \times M \cdot N$ to $B \times M \cdot N \times H$
  - ➔ and afterwards **reshape again** to have a $B \cdot M \cdot N \times H$ tensor
- Consequently we have to **reverse this** before returning the **output**

Convolutional Batch Normalization

- In CNNs batch normalization is adjusted to work **similar to convolution**
- A scalar μ , σ is calculated for the H **channels**
- Implementation can be reused, by observing
  - → that **spatial dimensions** M, N can be treated **like the batch dimension** B
  - → we can **reshape** the $B \times H \times M \times N$ tensor to $B \times H \times M \cdot N$
  - → because of our format we have to **transpose** from $B \times H \times M \cdot N$ to $B \times M \cdot N \times H$
  - → and afterwards **reshape again** to have a $B \cdot M \cdot N \times H$ tensor
- Consequently we have to **reverse this** before returning the **output**
- ... and do the **same** in the **backward pass**

LeNet (optional)

LeNet architecture

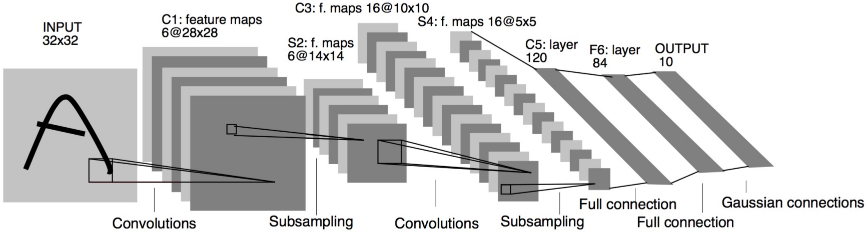

Figure: LeNet

Modified LeNet architecture

Deviations

- Input is 28×28
- Our conv only supports "same" padding - so C3 has **larger activation maps**
- Input to **C5** is also **larger**
- We only implemented ReLUs, so **no** TanH
- We also use the implemented SoftMax **instead of** RBF units

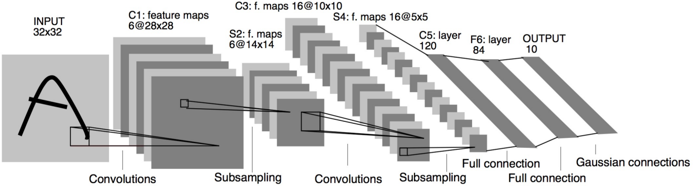

Figure: LeNet

Thanks for listening.

Any questions?