Regularization

K. Breininger, V. Christlein, Z. Yang, L. Rist, M. Nau, S. Jaganathan, C. Liu, N. Maul, L. Folle, M. Zinnen, K. Packhäuser
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
January 4, 2023

Tasks in this exercise

1. Optimization Constraints: Augmenting the loss function L1 and L2
2. Dropout **Layer**
3. Batch Normalization **Layer**

prominent regularization layers

4. LeNet: Put everything together **(optional)**
5. RNN layer: Elman Unit simplest recurrent layer
6. LSTM layer: Backpropagation at its best! **(optional)**

more suggested and the best example how to do the backpropagation

Optimization Constraints: Loss function augmentation

General outline

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

we contraint our network or weights that they contribute to total loss

we will punish the weight that the optimizer decide to make too large or punish too large weights because we don't want peaks

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

first apply the constraints
second do the adjustments of loss inside the neural network class

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

summing up the reg loss and add it
to the ordinary loss

- Constraints **only need** current weights

➔ Add constraint objects in the optimizer

because the optimizers have access
to the wights any way

- Since constraints generate part of the loss:

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

as only the weights are necessary to compute the constraints so add them to the optimizers
each layer instance has its own optimizer instance and thus constraints. so it is not reasonable
to integrate that to the loss function because we would need to have acces to all the weights
inside the loss function. so apply the constraints inside the optimizers

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:

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

- 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

it dos not have a forward pass and backward pass

- Forward pass:

landa : reg parameter

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

reg loss
for every layer we have
a different punishment or
reg term

- Backward pass:

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

adjustments in the optimizers
thtat's why we call it backward pass

partial derivatives
but because we
have L' we should
have adjustments

- 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.

L_1 regularization

- Forward pass:

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

punishment is different here

- Backward pass:

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

Dropout

Method

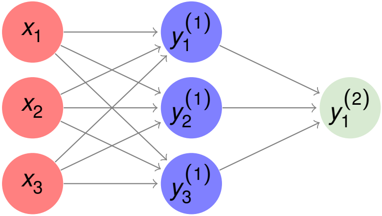

Figure: Dropout

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

Method

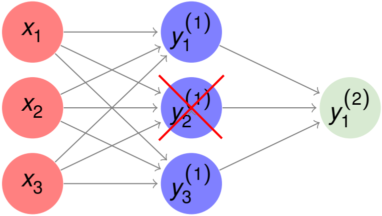

Figure: Dropout

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

Method

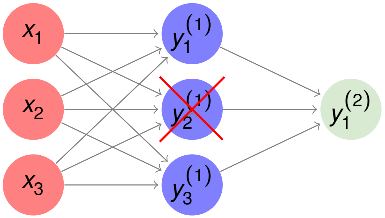

Figure: Dropout

you have set some signals to 0 you have silent some signals and reduced energy

- 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!

partial dervative

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Batch normalization

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 $\boldsymbol{\sigma}_B^2$ from **batch**

Forward pass

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

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

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ 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 $\boldsymbol{\sigma}_B^2$ from **batch**

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

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}^2$ 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 $\boldsymbol{\sigma}_B^2$ from **batch**

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

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}^2$ have the **same dimension** as the **input vectors**
- $\boldsymbol{\beta}$, γ and $\boldsymbol{\mu}_B$, $\boldsymbol{\sigma}_B^2$ have same **dimension** to be able to preserve **identity**

Forward pass

batch normalization between 2 layers

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

like bias and weights

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

variance

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ from **batch**

mioo and sigma are all vectors

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

gama and beta are vectors
multiplication is element wise

gama is the weight and beta is bias

- $\boldsymbol{\mu}$, $\boldsymbol{\sigma}^2$ have the **same dimension** as the **input vectors**
- $\boldsymbol{\beta}$, γ and $\boldsymbol{\mu}_B$, $\boldsymbol{\sigma}_B^2$ have same **dimension** to be able to preserve **identity**
- Notice that β is a **bias**

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ of the **training set**

Test time

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ 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 $\boldsymbol{\sigma}_B^2$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ of the **training set**
- It's **expensive** to calculate the true training set mean and variance
- Therefore a **moving average** is common:

$$\tilde{\boldsymbol{\mu}}^{(k)} \approx \alpha\tilde{\boldsymbol{\mu}}^{(k-1)} + (1-\alpha)\boldsymbol{\mu}_B^{(k)}$$
$$\tilde{\boldsymbol{\sigma}}^{\mathbf{2}(k)} \approx \alpha\tilde{\boldsymbol{\sigma}}^{\mathbf{2}(k-1)} + (1-\alpha)\boldsymbol{\sigma}_B^{\mathbf{2}(k)}$$

Test time

we dont have batch to have mioo and gama

- Test-time: replace $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ of the **training set**
- It's **expensive** to calculate the true training set mean and variance
- Therefore a **moving average** is common:

moving mioo of the previous

current iteration

like momentum

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

$$\tilde{\boldsymbol{\sigma}}^{\mathbf{2}(k)} \approx \alpha \tilde{\boldsymbol{\sigma}}^{\mathbf{2}(k-1)} + (1-\alpha)\boldsymbol{\sigma}_B^{\mathbf{2}(k)}$$

we keep updating them

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

like mmentum

- The exponent (k) and (k-1) are iteration-indices!

batch normalisation is used inside the neural network. the network changes each time. we augment the data each time so we can't hardcode it

Backward pass

we want to optimise gama and beta

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

error tensor times input over batches

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

so we need gradients with respect to the gama

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

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

gradients with respect to the beta

Backward pass

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

loss with respect to the output `= error tensor

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

element wise multiplication with weights

sum over batch

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

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

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

we need these gradients for the loss gradients

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
  - rescale → we can **reshape** the $B \times H \times M \times N$ tensor to $B \times H \times M \cdot N$ flatten last height and width
  - → because of our format we have to **transpose** from $B \times H \times M \cdot N$ to $B \times M \cdot N \times H$ batch . channels. height. width
    transpose
  - → 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**

we reshape or flatten the first 2 dimensions

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

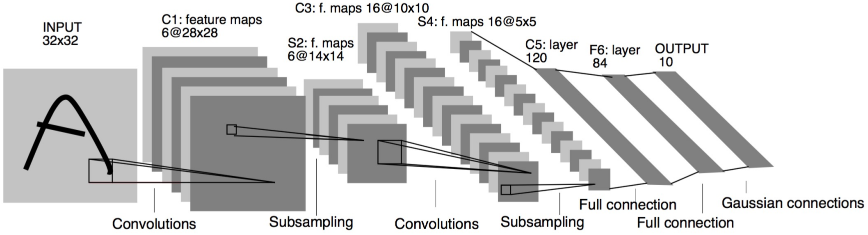

Figure: LeNet

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

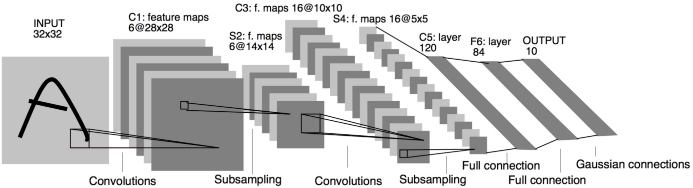

Figure: LeNet

Thanks for listening.

Any questions?