Lecture 7

Layer norm, FRN, TLU, Introduction to Keras

Normalization in CNNs

- **Batch Normalization:** Standardizes activations within a feature channel to have zero mean and unit variance, facilitating training and enabling larger learning rates.

- **Challenges:** Struggles with small batch sizes due to unreliable estimated mean and variance parameters.

Data Representations

- **Tensor in CNNs:** Remember that in CNNs, data is represented in the form of a tensor.

- A specific element in the tensor can be accessed using an index

  $$i = (i_N, i_H, i_W, i_C)$$

  - **(i_N): Batch Size,** Number of examples processed together in a single forward/backward pass.
  - **(i_H) (i_W): Height and Width,** Spatial dimensions of the input image or feature map, representing the vertical and horizontal pixels respectively.
  - **(i_C): Channels,** Depth of the input image or feature map, representing color channels (e.g., RGB) or feature channels in deeper layers.

Layer Normalization

Computes mean and variance across dimensions (height, width, channel) but not across batch examples.

$$\mu_i = \frac{1}{|S_i|} \sum_{k \in S_i} z_k$$

$$\sigma_i = \sqrt{\frac{1}{|S_i|} \sum_{k \in S_i} (z_k - \mu_i)^2 + \epsilon}$$

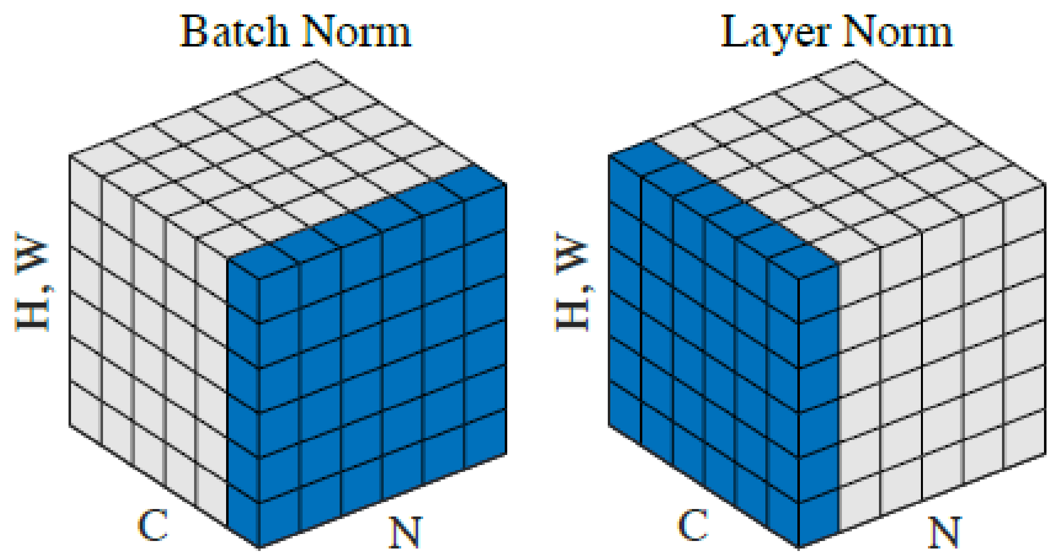
- μ_i and σ_i: Computed mean and standard deviation.
- S_i: Set of elements pooled across, determined by the specific normalization technique and dimensions being normalized.
- z_k: Individual data points in the tensor, such as activation values.
- ϵ: Small constant for numerical stability.

Illustration of different normalization methods

- The pixels in blue are normalized by the same mean and variance

- In Batch Norm, we pool over batch, height, width,

- In Layer Norm we pool over channel, height and width



Credit: Y. Wu and K. He. “Group Normalization”. In: ECCV. 2018.

Filter Response Normalization (FRN)

- S. Singh and S. Krishnan. “Filter Response Normalization Layer: Eliminating Batch Dependence in the Training of Deep Neural Networks”. In: CVPR. 2020.
- Robust normalization technique with small batch sizes

Filter Response Normalization (FRN)

Given an input z for a specific channel and batch entry

$$\hat{z} = \frac{z}{\sqrt{\bar{z}^2 + \epsilon}}$$

Where:

$$\bar{z}^2 = \frac{1}{N} \sum_{i,j} z_{bijc}^2$$

And ϵ is a small constant to avoid division by zero.

Scaling and Shifting with TLU

Post-normalization, the activations are scaled and shifted using learnable parameters, followed by the application of the Thresholded Linear Unit (TLU):

$$\tilde{z} = \gamma \hat{z} + \beta$$

$$y = \max(x, \theta)$$

Where θ is a learnable parameter ensuring the activations are bounded below, thus mitigating the "dying ReLU" problem.

Filter Response Normalization (FRN)

- **Stability:** Provides stable activations and gradients.
- **Robust Training:** Ensures consistent and robust training across various batch sizes and network architectures.

Normalizer-free networks

- A. Brock, S. De, S. L. Smith, and K. Simonyan. "High-Performance Large-Scale Image Recognition Without Normalization". In (2021). arXiv: 2102.06171 [cs.CV].

- A methodology that trains deep residual networks without utilizing batch normalization or other normalization layers.

- Adaptive Gradient Clipping, which dynamically adjusts the clipping strength during training to avoid instabilities.

Gradient Clipping

Gradient Clipping: prevent gradients from becoming too large.
if g is the gradient, and c is a predefined clipping threshold, the clipped gradient g' is calculated as:

$$g' = \min\left(1, \frac{c}{\|g\|}\right) g$$

Ensures the norm of the gradient never exceeds a specified limit, while maintaining its direction.

Dynamic Gradient Clipping in Normalizer-Free Networks

c is dynamic.

$$c_t = \alpha \cdot \text{mean}\left(\|g_{t-1}\|\right) + \beta$$

Where:

- c_t is the dynamic clipping threshold at time t.
- α and β are hyperparameters controlling adaptation speed and base level of c, respectively.
- g_{t-1} represents the gradient at the previous timestep $t-1$.
- The mean function calculates the average norm of the gradient across mini-batches.

Common architectures for image classification

- AlexNet 2012
- GoogLeNet (Inception) 2015
- ResNet 2015
- DenseNet 2017
- ConvNet 2022 (Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. “A ConvNet for the 2020s”. In: (2022). arXiv: 2201.03545 [cs.CV].)

Neural Architecture Search (NAS)

- **NAS:** Automated neural network design.
- **Goal:** Optimize architecture for specific tasks.

Challenges and Objectives in NAS

- optimize for various objectives (accuracy, model size, etc) simultaneously,
- The primary challenge in NAS is the computational expense of evaluating the objective, which involves training each candidate model.
- Solutions include using Bayesian optimization to reduce calls to the objective function, creating differentiable approximations to the loss, and converting the architecture into a kernel function.

NAS approaches

- **Bayesian Optimization:** Reduces the number of calls to the objective function.
- **Differentiable Approximations:** Allows the use of gradient-based optimization methods to navigate the architecture search space.
- **Neural Tangent Kernel Method:** Converts the architecture into a kernel function, enabling the analysis of its eigenvalues to predict performance without actual training.

Keras: The Python Deep Learning library

Some slides courtesy of Aref Jafari

Step 1) Import Libraries

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, Input
from keras.utils import np_utils

#Other types of layers
from keras.layers import LSTM
from keras.layers import Conv1D, Conv2D, Conv3D, MaxPooling2D

from keras.layers.normalization import BatchNormalization

import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(2017)
```

Step 3) Define model architecture

Form 1)

In [11]:
```
model = Sequential()
model.add(Dense(512, activation='relu', use_bias=True,  input_shape=(784,)))
model.add(Dense(128, activation='relu', use_bias=True))
model.add(Dense(10, activation='softmax', use_bias=True))
```

Form 2)

In [91]:
```
from keras.models import Model

X_inp = Input(shape=(784,))
h1 = Dense(512, activation='relu', use_bias=True)(X_inp)
h2 = Dense(128, activation='relu', use_bias=True)(h1)
h3 = Dense(10, activation='softmax', use_bias=True)(h2)

model = Model(inputs=X_inp, outputs=h3)
```

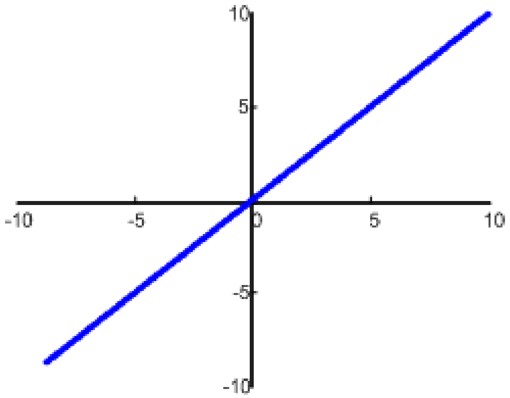
Step 3) Define model architecture
(Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

linear

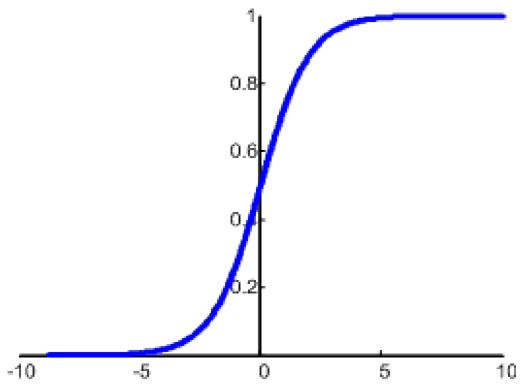


$$f(x) = x$$

Step 3) Define model architecture
(Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

sigmoid

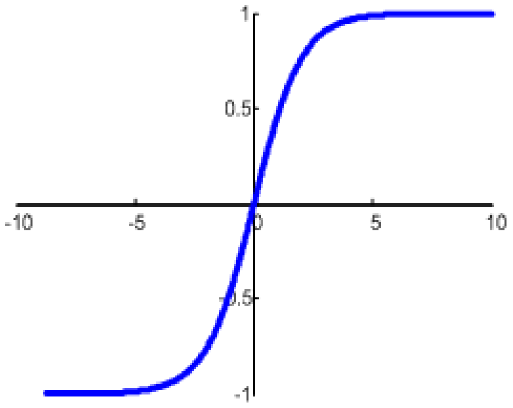


$$f(x) = \frac{1}{1 - e^{-x}}$$

Step 3) Define model architecture
(Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

tanh

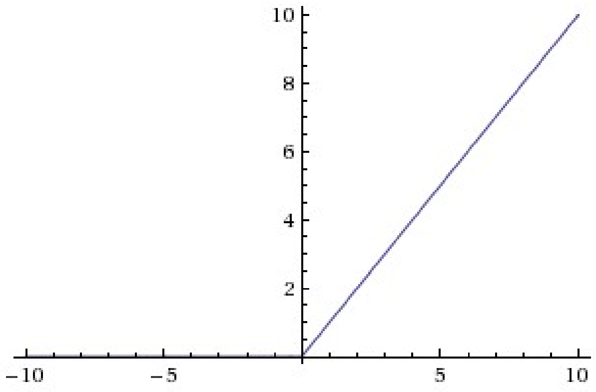


$$f(x) = \tanh(x)$$

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

relu

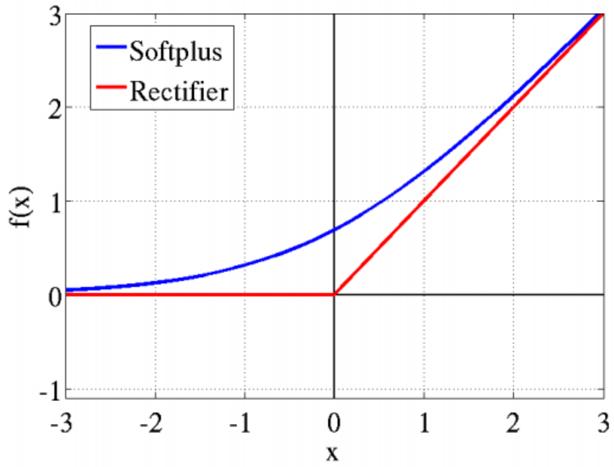


$$f(x) = \max(0, x)$$

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

softplus

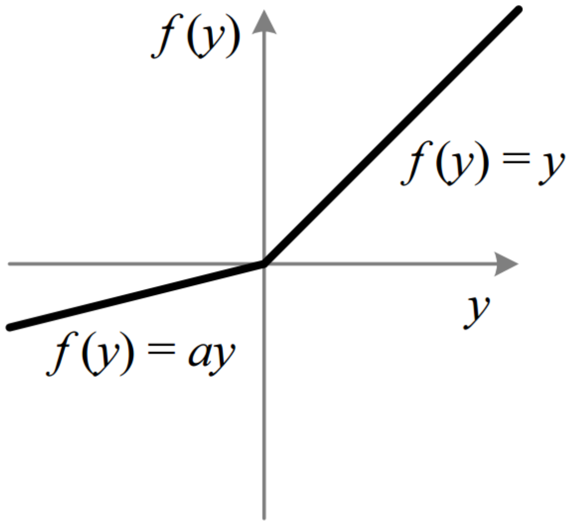


$$f(x) = \ln[1 + e^x]$$

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

LeakyReLU



Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \; for\; j = 1, \ldots, K$$

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

Step 3) Define model architecture
(Other attributes of Dense layer)

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='zeros',
                        kernel_regularizer=None,
                        bias_regularizer=None,
                        activity_regularizer=None,
                        kernel_constraint=None,
                        bias_constraint=None)
```

Step 3) Define model architecture (Other attributes of Dense layer)

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='zeros',
                        kernel_regularizer=None,
                        bias_regularizer=None,
                        activity_regularizer=None,
                        kernel_constraint=None,
                        bias_constraint=None)
```

instances of
keras.regularizers.Regularizer
(l1, l2, …)

Step 3) Define model architecture (Other attributes of Dense layer)

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='zeros',
                        kernel_regularizer=None,
                        bias_regularizer=None,
                        activity_regularizer=None,
                        kernel_constraint=None,
                        bias_constraint=None)
```

Functions from the **constraints** module allow setting constraints (eg. non-negativity) on network parameters during optimization

Example:

```
from keras.constraints import maxnorm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

Available constraints

max_norm(max_value=2, axis=0): maximum-norm constraint

non_neg(): non-negativity constraint

unit_norm(): unit-norm constraint, enforces the matrix to have unit norm along the last axis

Step 3) Define model architecture (Other attributes of Dense layer)

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='zeros',
                        kernel_regularizer=None,
                        bias_regularizer=None,
                        activity_regularizer=None,
                        kernel_constraint=None,
                        bias_constraint=None)
```

Step 3) Define model architecture (Dropout Layers)

```
keras.layers.core.Dropout(rate,
                          noise_shape=None,
                          seed=None)
```

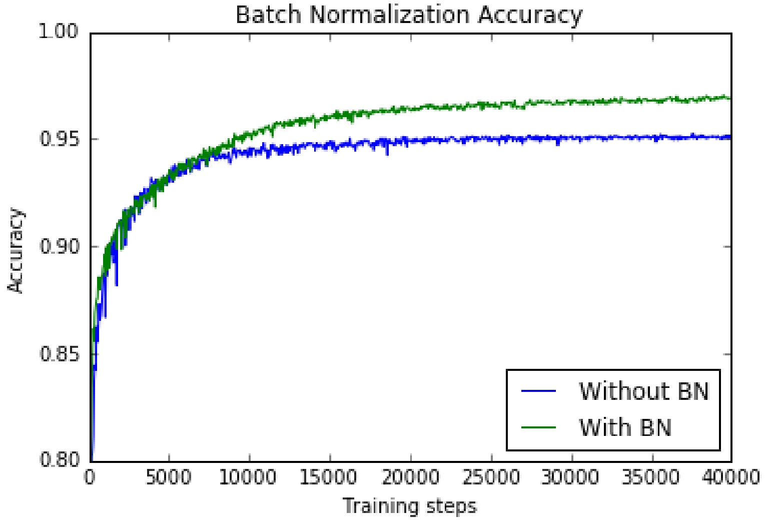
Example:

```
model.add(Dense(128, activation='relu', use_bias=True))
model.add(Dropout(0.2))
```

Step 3) Define model architecture (Batch Normalization Layers)

Example:

```
model = Sequential()
model.add(Dense(64, input_dim=14))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
```



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Step 4) Compile model
(Loss functions)

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

Available loss functions:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Step 4) Compile model (Loss functions)

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

Custom loss function

```
import theano.tensor as T

def myLoss(y_true, y_pred):
    cce = T.mean(T.sqr(y_true-y_pred))
    return cce
```

```
model.compile(optimizer='adadelta', loss=myLoss)
```

Available loss functions:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Step 4) Compile model (Optimizers)

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

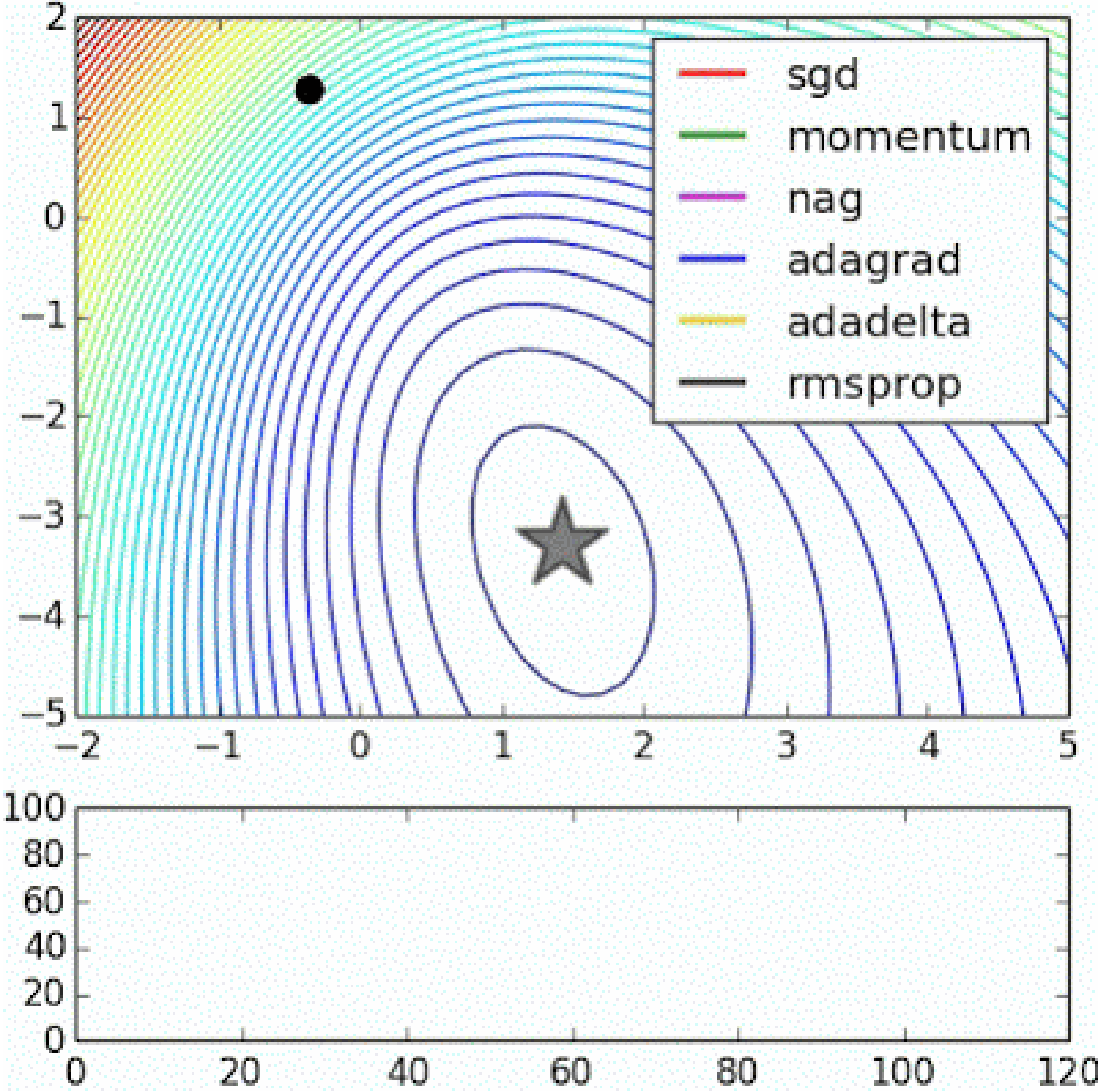
Available loss functions:

- SGD
- RMSprop
- Adagrad
- Adadelta
- Adam
- Adamax
- Nadam
- TFOptimizer

Adagrad

```
adagrad = keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
```

```
model.compile(optimizer=adagrad, loss=myLoss)
```


sgd
momentum
nag
adagrad
adadelta
rmsprop
2
1
0
−1
−2
−3
−4
−5
−2
−1
0
1
2
3
4
5
100
80
60
40
20
0
0
20
40
60
80
100
120