Deep learning, Basics

Anders Eklund

anders.eklund@liu.se

Department of Biomedical Engineering (IMT)
Department of Computer and Information Science (IDA)
Center for Medical Image Science and Visualization (CMIV)
Linköping University, Sweden

April 1, 2022

Outline

- Deep learning basics
  - Activation functions (Section 6.3.1)
  - Regularization techniques (Chapter 7)
  - Working with (mini)batches
  - Computer hardware
  - Optimization methods (Chapter 8)
  - Initialization of weights (Section 8.4)
  - Batch normalization (Section 8.7.1)

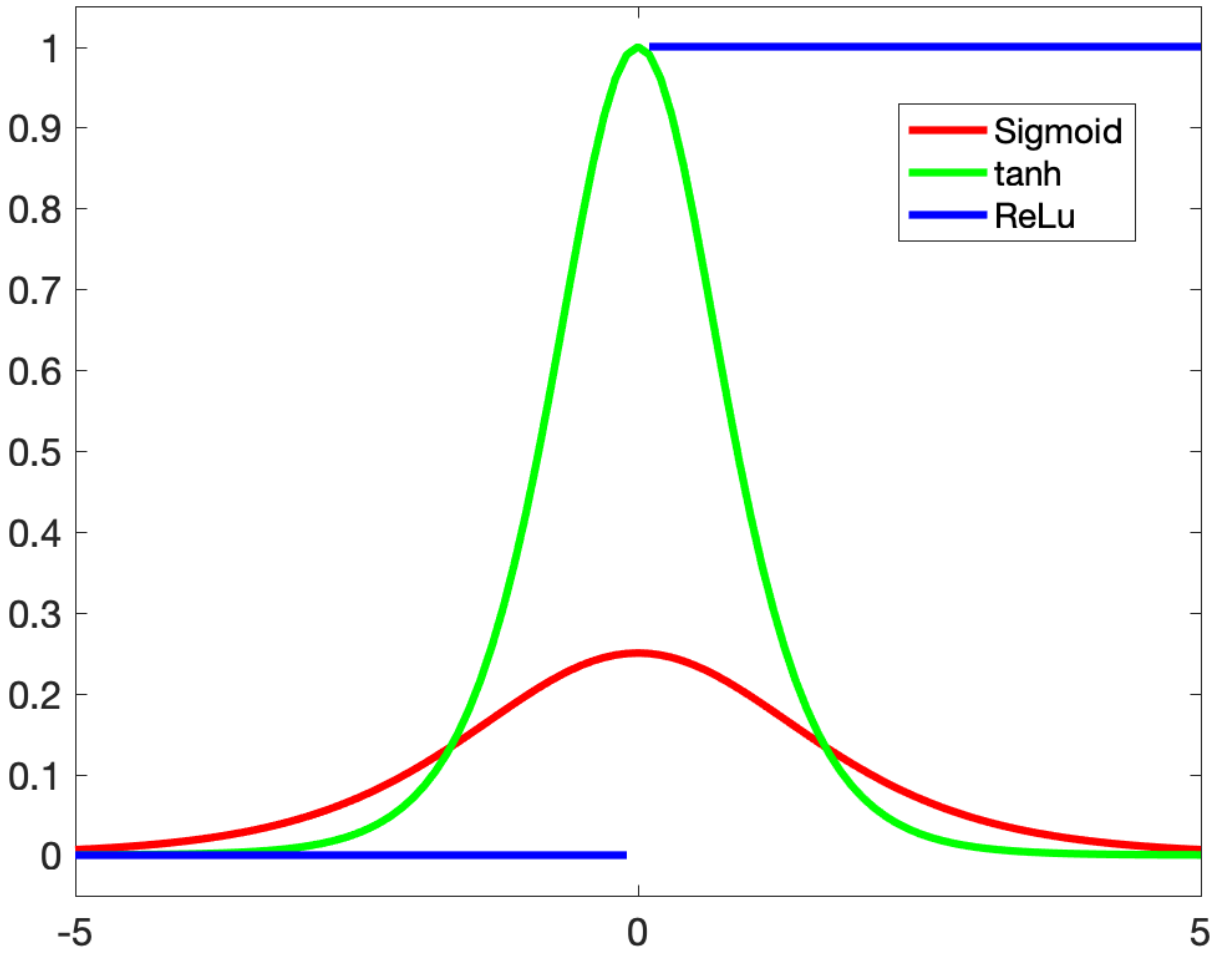
Activation functions

- What happens if all activation functions in a network are linear?

Activation functions

- Before deep learning, most neural networks used sigmoid or hyperbolic tangent as activation functions
- In deep learning,
  ReLU (rectified linear unit) and Leaky ReLU are most common
  (many other ReLU variants exist)
- ReLU: $y = max(0, x)$
- For ReLU, the gradient is always large for positive activations
  (i.e. no vanishing gradients), but 0 for negative activations

Activation functions - gradients



For ReLU, the gradient is never zero for activations > 0

Gradient of ReLU

- But ReLU is not differentiable at 0?
- Can gradient based learning be used with ReLU?
- Not a problem in practice, we usually never reach a local minimum (gradient is exactly 0)
- Stochastic gradient descent -> noisy gradient

Why ReLU?

- ReLU is almost linear (but still non-linear), easy to optimize, the gradient is always large for positive activations
- The second derivative is 0 almost everywhere, i.e. the gradients are consistent
- Drawback: gradient is 0 if activation is negative

ReLU vs parametric ReLU

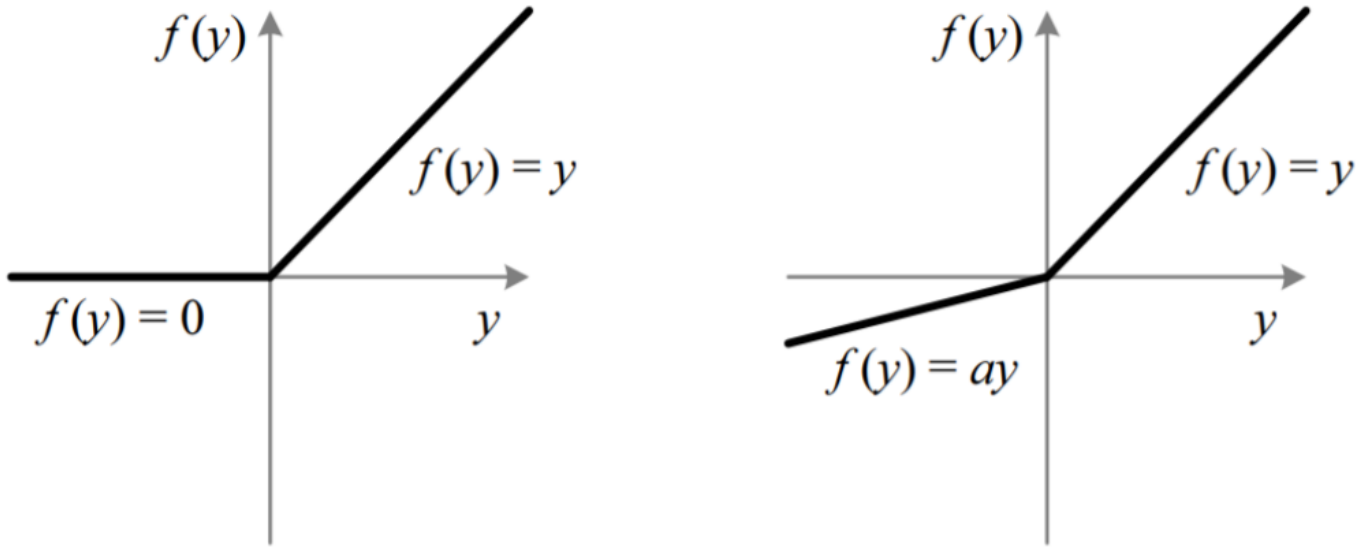


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. IEEE international conference on computer vision (pp. 1026-1034).

Leaky vs parametric ReLU

- Leaky ReLU: αx for $x < 0$, x for $x >= 0$, hyperparameter α
- For Leaky Relu, the gradient is only 0 if $x = 0$
- What is the difference between leaky and parametric ReLU?
- Leaky ReLU: set α to small value like 0.01
- Parametric ReLU: treat α as a parameter to learn

Leaky ReLU in Keras

relu

```
keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0.0)
```

Rectified Linear Unit.

With default values, it returns element-wise `max(x, 0)`.

Otherwise, it follows: `f(x) = max_value` for `x >= max_value`, `f(x) = x` for `threshold <= x < max_value`, `f(x) = alpha * (x - threshold)` otherwise.

Arguments

- **x**: Input tensor.
- **alpha**: float. Slope of the negative part. Defaults to zero.
- **max_value**: float. Saturation threshold.
- **threshold**: float. Threshold value for thresholded activation.

Regularization techniques

- To obtain state-of-the-art results,
  deep learning uses deep networks
  with millions (billions) of parameters
- This increases the risk of overfitting,
  regularization is very important in deep learning
- The best regularization is always to have
  more training data, but not always possible

Regularization techniques

- The performance is often high on the training set,
  but lower on validation and testing set
- Different regularization techniques are available
  to prevent overfitting to the training data
  (reduce generalization error without changing training error)
- Find the best weights, subject to different penalties
- Prefer networks with few and small weights
- Start with large complex network,
  using regularization will find the "best" sub-network

Regularization techniques - L1 & L2

- L1 norm (LASSO),

  $\lambda_1 \sum_i |w_i|$

- L2 norm (ridge regression / Tikhonov),

  $\lambda_2 \sum_i w_i^2$

- L1 + L2 norm (elastic net),

  $\lambda_1 \sum_i |w_i| + \lambda_2 \sum_i w_i^2$

- Add penalty term to loss function, e.g.

  $L = \sum_{\mu=1}^{N} -(t^\mu log y^\mu + (1 - t^\mu) log(1 - y^\mu)) + \lambda_2 \sum_i w_i^2$

L2 REGULARIZATION

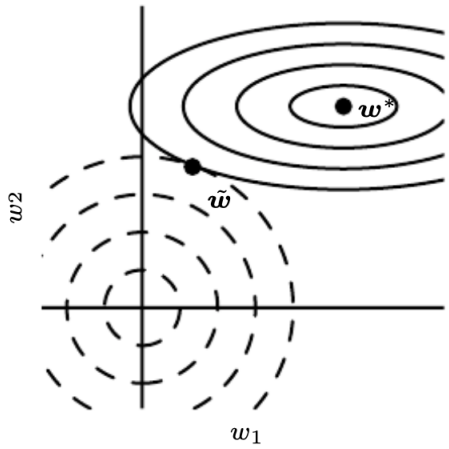


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

Important weights will not be changed much.
Unimportant weights are pulled towards 0.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Adding L2 regularization - Keras

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation
from keras.regularizers import l2

model = Sequential()

model.add(Dense(32, activation='sigmoid',
                input_dim=20))
model.add(Dense(64, kernel_regularizer=l2(0.01),
                    bias_regularizer=l2(0.02) ))
model.add(Dense(1, activation='sigmoid'))
```

Same principle for L1, and L1 + L2, see https://keras.io/regularizers/

The gradient will automatically include the regularization term

L1 REGULARIZATION - GRADIENT

- Without regularization

  $W = W - \epsilon \nabla L(W)$

- With L_2 regularization

  $W = W - \epsilon \nabla L(W) - \epsilon \lambda_2 W$

- With L_1 regularization

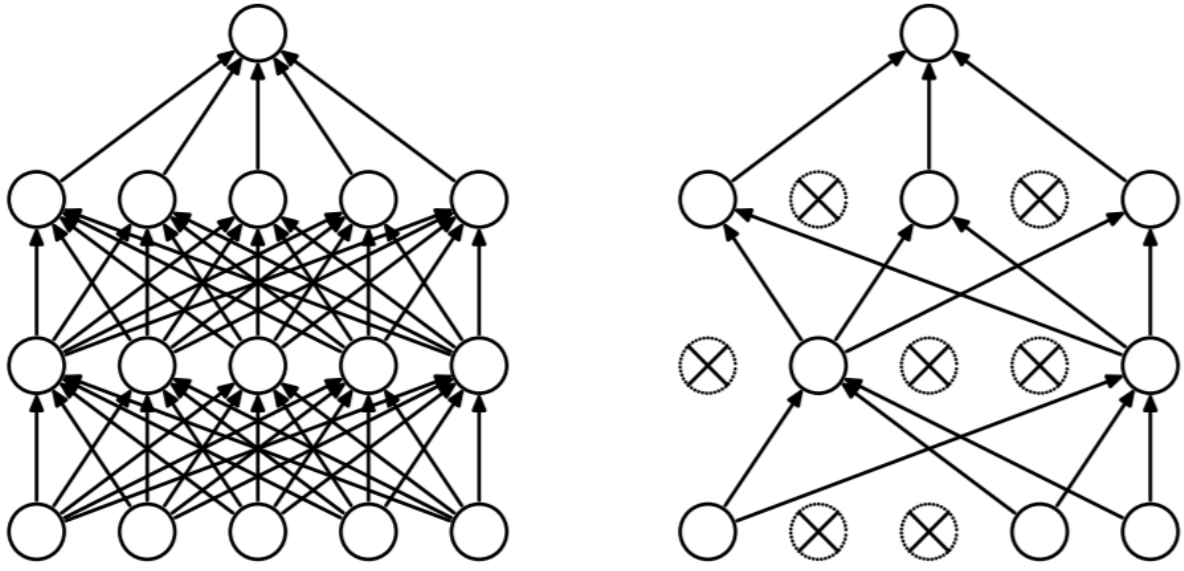
  $W = W - \epsilon \nabla L(W) - \epsilon \lambda_1 sign(W)$

- L_1: Shrink the weight vector by constant factor for each dimension (no linear scaling with w)

Regularization techniques - dropout

- Dropout - randomly set hidden units to 0
  in a fully connected (dense) layer during training,
  with a given probability (e.g. 50%)
- Network cannot rely on a few connections,
  makes it better at generalizing to new data
- A way to combine many different neural network architectures
- Dropout is normally only used for training, not for testing,
  but can be used as simple way of getting uncertainty
  (Monte Carlo dropout, used in Lab 1)

REGULARIZATION TECHNIQUES - DROPOUT



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

Regularization techniques - dropout

- Dropout will have an effect similar to L_2 regularization
- Since nodes can be removed randomly,
  training cannot put a high weight on a few nodes,
  need to put smaller weights on many nodes
- The training becomes more irregular,
  since loss function is slightly different every iteration
  (more training epochs may be required)
- Can have different dropout rates for different layers
  - Should use higher dropout rate in layers with many weights
    (higher risk of overfitting)
  - Not necessary to use dropout in layers with few weights
    (low risk of overfitting)

Adding dropout regularization - Keras

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Dropout

model = Sequential()

model.add(Dense(32, activation='sigmoid',
                input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

See https://keras.io/api/layers/regularization_layers/dropout/

Regularization techniques - indirect

- Data augmentation
  - Increase the amount of realistic training data,
    by modifications of the existing training data
    (Lecture 4), easy for image data, maybe not for other data
- Early stopping
  - Stop training when validation error does not improve
- Bagging / Ensemble
  - Train several models separately (different random initialization),
    let all models vote (time consuming)
- Multi-task learning
  - Train a feature extractor to be good at several tasks
    (e.g. classification + segmentation)

MULTI-TASK LEARNING - EXAMPLE

- Combined loss function

$$L = L_{segmentation} + 0.1L_{autoencoder} + 0.1L_{KL-divergence}$$

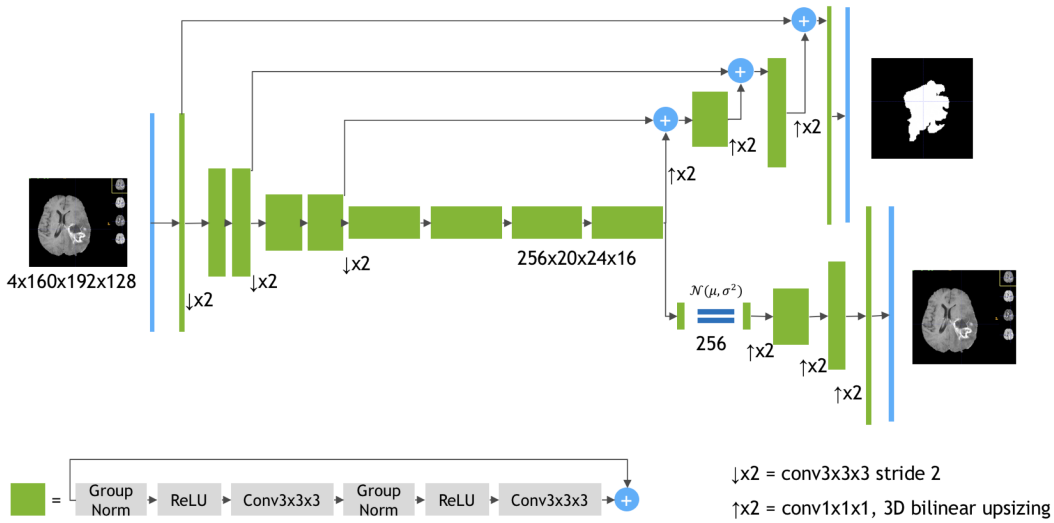


Fig. 1. Schematic visualization of the network architecture. Input is a four channel 3D MRI crop, followed by initial 3x3x3 3D convolution with 32 filters. Each green block is a ResNet-like block with the GroupNorm normalization. The output of the segmentation decoder has three channels (with the same spatial size as the input) followed by a sigmoid for segmentation maps of the three tumor subregions (WT, TC, ET). The VAE branch reconstructs the input image into itself, and is used only during training to regularize the shared encoder.

- Myronenko, A. (2018). 3D MRI brain tumor segmentation using autoencoder regularization. In International MICCAI Brainlesion Workshop (pp. 311-320). Springer

EARLY STOPPING

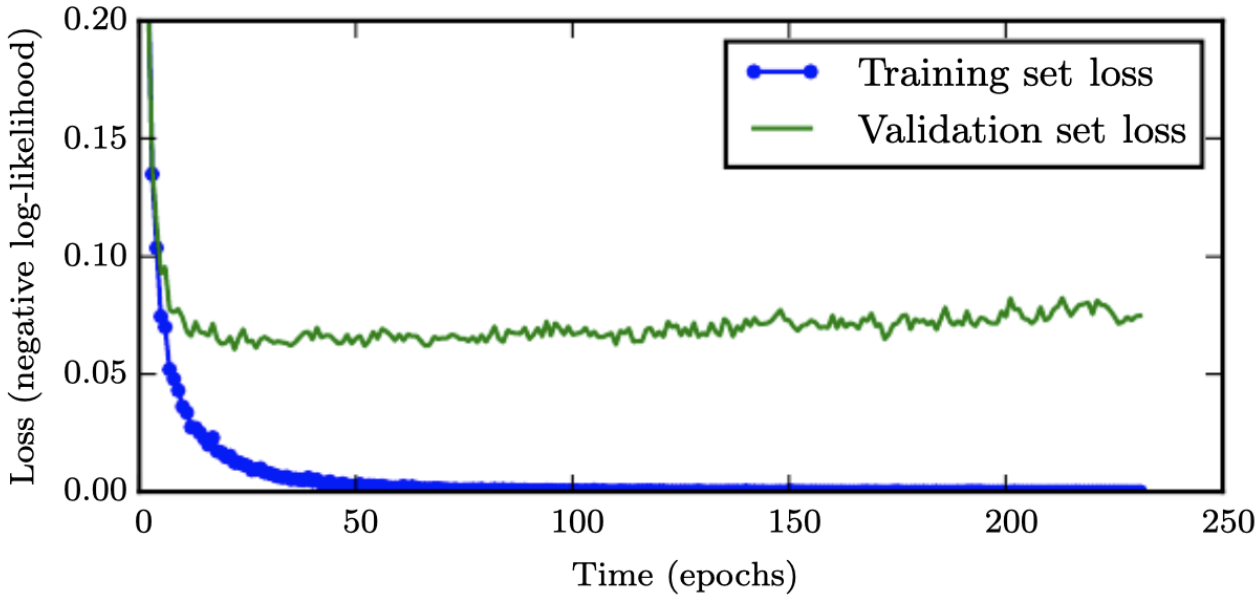


Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Early stopping - Keras

```
from keras.callbacks import EarlyStopping

# Simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min')

# Patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min',
patience=50)

model.fit(Xtrain, Ytrain, validation_data=(Xval, Yval)
callbacks=[es])
```

Requires that you specify validation data

EARLY STOPPING - KERAS

EarlyStopping [source]

```
keras.callbacks.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0,
```

Stop training when a monitored quantity has stopped improving.

Arguments

- **monitor**: quantity to be monitored.
- **min_delta**: minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience**: number of epochs that produced the monitored quantity with no improvement after which training will be stopped. Validation quantities may not be produced for every epoch, if the validation frequency (`model.fit(validation_freq=5)`) is greater than one.
- **verbose**: verbosity mode.
- **mode**: one of {auto, min, max}. In `min` mode, training will stop when the quantity monitored has stopped decreasing; in `max` mode it will stop when the quantity monitored has stopped increasing; in `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **baseline**: Baseline value for the monitored quantity to reach. Training will stop if the model doesn't show improvement over the baseline.
- **restore_best_weights**: whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used.

Optimization methods

- Since deep networks often have millions of parameters, the optimization methods become more important
- Normally not possible to fit all data into GPU memory (e.g. 8 - 32 GB) have to work on batches of data
- SGD, stochastic gradient descent, gradient descent on (mini)batches
  - Benefit: Can work with large datasets (e.g. batch size 32, 10,000 training samples)
  - Drawback: Gradient is more noisy (uncertain)
  - Necessary to reduce learning rate over time (due to sampling noise)

Working with (mini)batches

- Can normally not fit all training data in GPU memory
- Also need to store weights of network,
  outputs / filter responses for each layer,
  gradients of all weights
- Need to work on batches of a smaller number of training examples,
  in some cases batch size is 1 (e.g. single image / volume)
- Most extreme case, cannot fit a single training example on GPU,
  digital pathology; image can be 50 000 x 50 000 pixels,
  need to work on patches / subvolumes
- Problem with small batches: gradient is more noisy
  (but this noise can have a regularization effect)

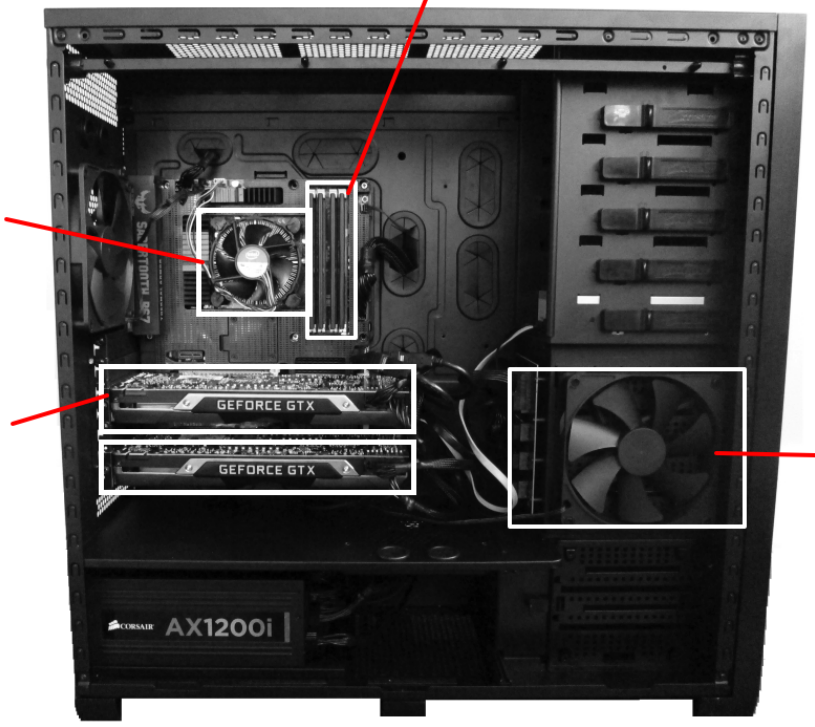
COMPUTER HARDWARE

CPU memory (RAM) 4 - 128 GB

Fast memory

CPU

4 - 20 cores

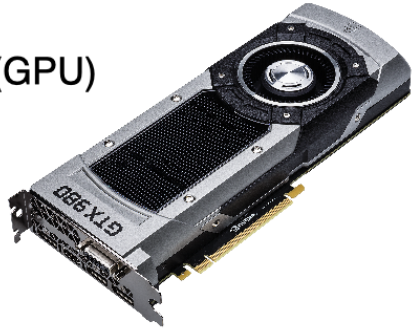


Graphics card (GPU)

2048 cores

2 - 32 GB

Superfast memory

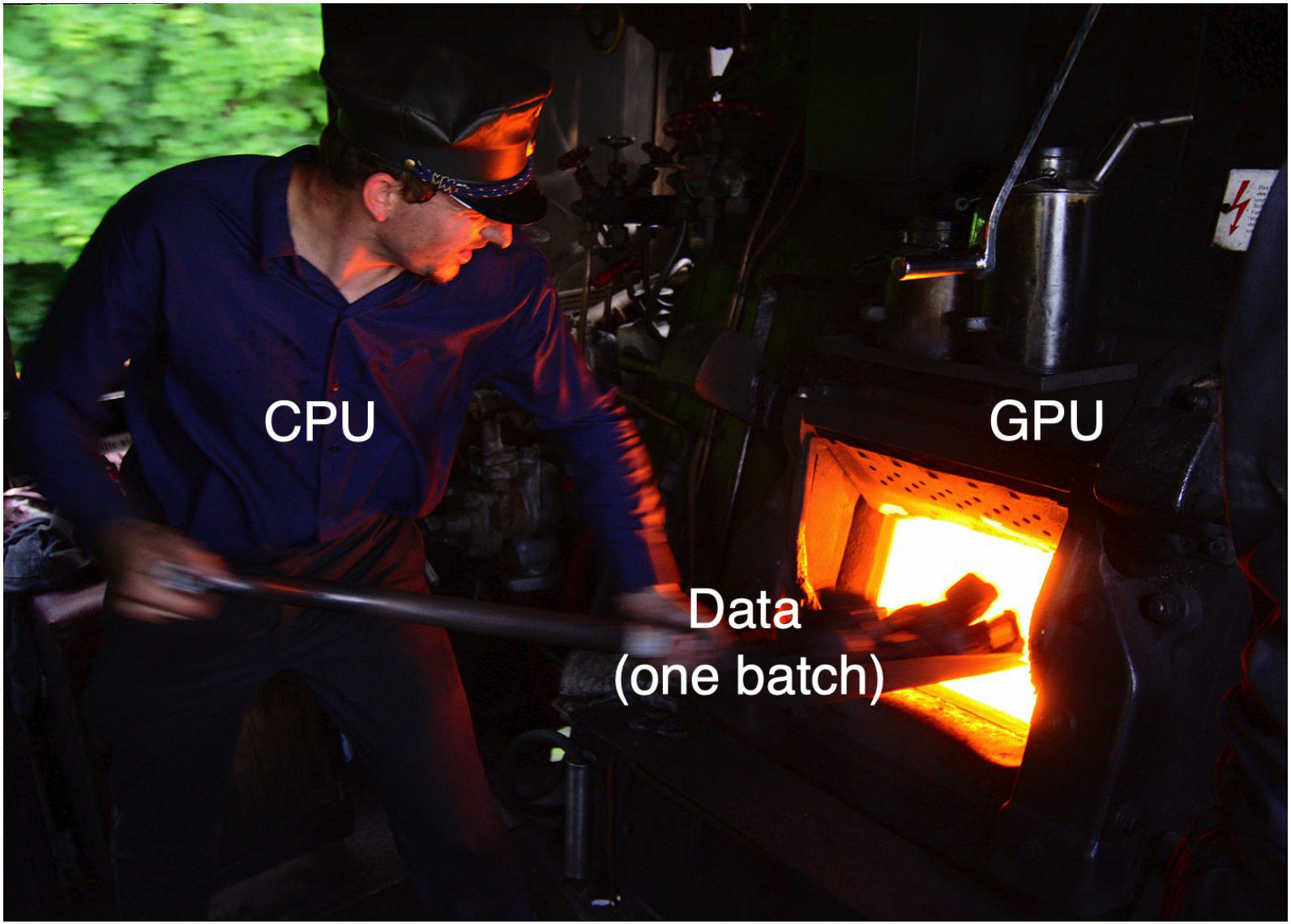


Hard drive

(2 - 20 TB)

Slow memory

Feeding a GPU with data



By Mussklprozz - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=15503515

Computer hardware

- If your data can fit in CPU memory (RAM),
  load all data into CPU memory,
  the CPU will then quickly feed the GPU with batches
- If your data does not fit in CPU memory,
  need to stream data from the hard drive (or network)
  (e.g. Keras ImageDataGenerator,
  https://keras.io/preprocessing/image/#imagedatagenerator-class)
- For convolutional neural networks (CNNs),
  common that the CPU performs on-the-fly image augmentation
  (rotations, scale, etc) before feeding each batch to the GPU

Training with generator

```
# All data stored in RAM, no augmentation
model.fit(Xtrain, Ytrain, batch_size=100,
          epochs=100, verbose=2,
          validation_data=(Xval, Yval))

# Data generated by generator (from data in RAM)
model.fit_generator(generator.flow(Xtrain, Ytrain,
        batch_size=100), validation_data=(Xval, Yval),
        steps_per_epoch=len(Xtrain) // 100, epochs=100
```

Training with generator

```
from keras.preprocessing.image import
ImageDataGenerator

# Define generator, including augmentation
generator = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise
# normalization (std, mean, and principal components
# if ZCA whitening is applied)
generator.fit(Xtrain)
```

Training with generator

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

Training with generator

```
validation_generator = test_datagen.flow_from_directory
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        steps_per_epoch=2000,
        epochs=50,
        validation_data=validation_generator,
        validation_steps=800)
```

Training with generator

- 'flow_from_directory' will read images from a directory, (PNG, JPG, ...), resize them to the target size
- The CPU performs the image augmentation
- Possible to save the augmented images to a directory
- Possible to create your own generator, to fit your specific data
- Note that streaming from the hard drive may be faster if you have a SSD instead of mechanical drive

Training with generator

- Training with model.fit; show the same images during each epoch
- Training with on-the-fly augmentation; means that we use different training data (e.g. images) during each batch / epoch
- Question: How will this affect how quickly the training accuracy improves?

Batch size and learning rate

- Batch size and learning rate are closely related
- Low batch size, might need smaller learning rate,
  as gradient is more uncertain
- Large batch size, might need larger learning rate,
  due to fewer weight updates in total
- Number of weight updates per epoch =
  number of training examples / batch size
- To increase number of weight updates,
  with constant batch size, increase the number of epochs

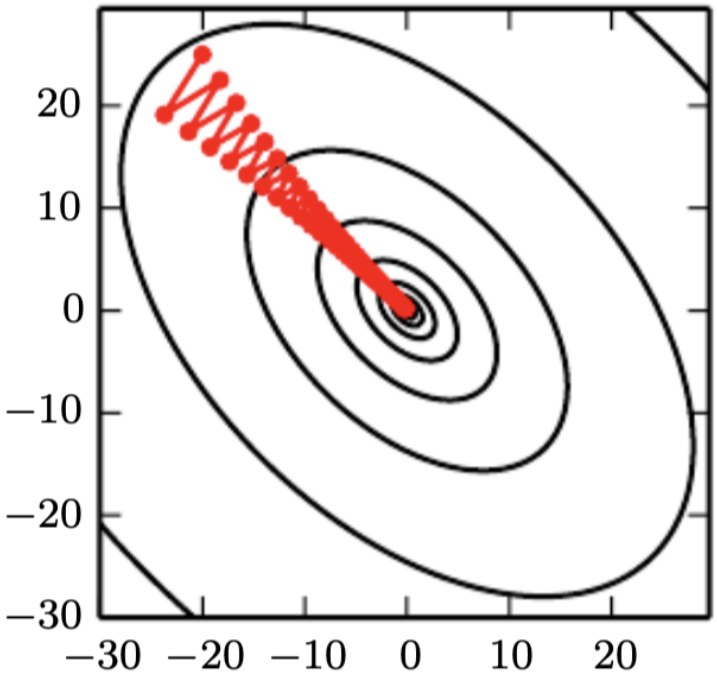
Batch size

- How large should the batch size be?
- Closely related to the amount of memory on the graphics card
- Too large batch size => training will crash
- Too small batch size => graphics card is not working 100%
  (i.e. your training is taking longer than it needs to)
- Use commands like "nvidia-smi" to see how busy the graphics card is
  (note that these commands sometimes show that 100% of the
  memory is used, even though only a small part of the memory is used)

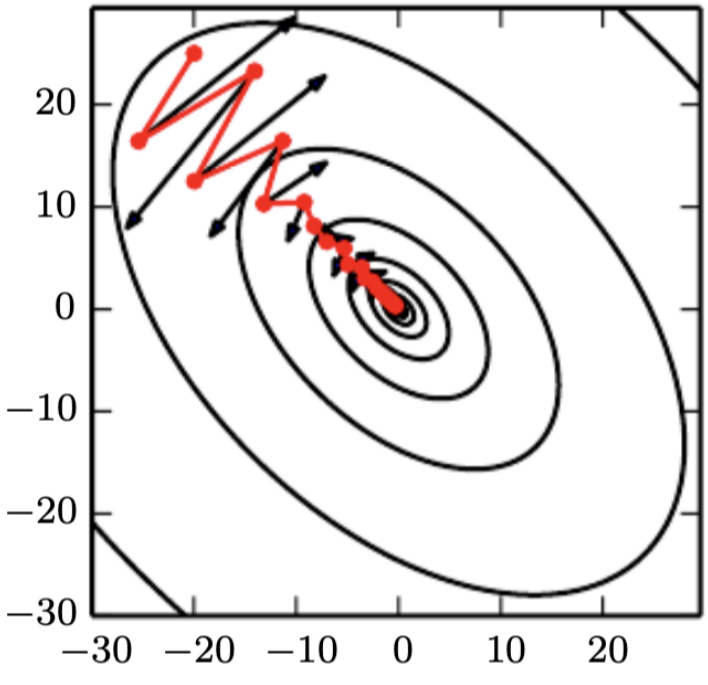
Optimization methods - momentum

- SGD can be rather slow, add momentum term
- Smooth the gradient by updating the weights
  using a combination of current and past gradients
- Faster convergence,
  larger step size if the gradient is similar in many updates
- Velocity update, $v = \alpha v - \epsilon \nabla L(W)$
- Update parameters, $W = W + v$
- Hyperparameter α, e.g. 0.9
- Also see Nesterov formula

Optimization methods - momentum



SGD



Momentum

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Optimization methods

- Question from lecture 1: Why not use second derivative?

Optimization methods

- Question from lecture 1: Why not use second derivative?
- Networks may have millions of parameters,
  difficult to use methods that involve working
  with matrices (e.g. Hessian) that contain all parameters
- Network with 20 million parameters =>
  Hessian matrix is 20 million x 20 million
- 1600 TB of memory to store Hessian matrix in 32 bit floats!
- (GPT-3: text model with 175 BILLION parameters)
- Methods not needing the Hessian (e.g. BFGS)

Optimization methods - adagrad

- Adapt the learning rate of all weights in model (instead of fix)
- Updates in any direction are made with the same proportions (down-scale steep gradients, upscale small gradients)
- Prevent gradients from being too large or too small
- Accumulate squared gradient, $r = r + \nabla L(W)^2$
- Update parameters, $W = W - \epsilon \frac{\nabla L(W)}{\sqrt{r}+\delta}$
- δ is a small constant, for numerical stability
- Division and square root applied element wise

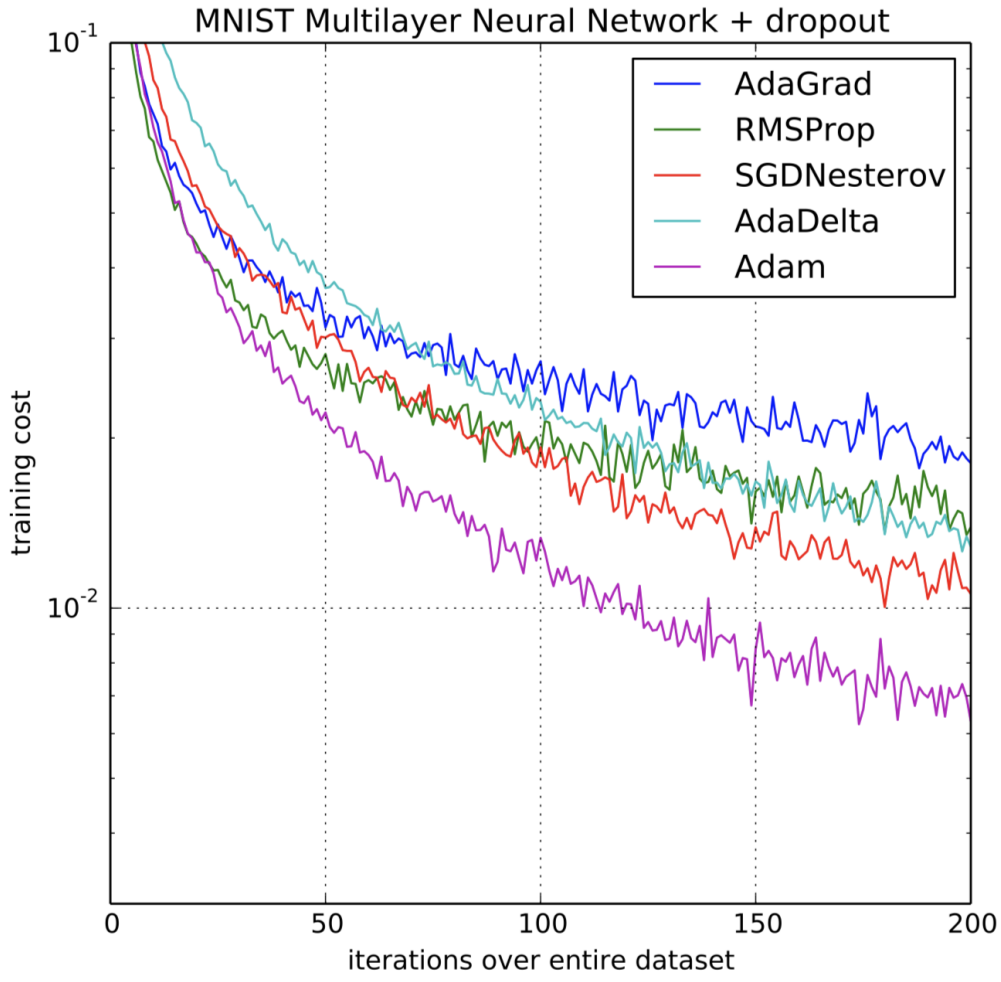
Optimization methods - RMSprop

- Similar to AdaGrad, but uses a weighted moving average instead of summing all gradients
  (learning rate will become too low for AdaGrad)
- Accumulate squared gradient, $r = \rho r + (1 - \rho) \nabla L(W)^2$
- Update parameters, $W = W - \epsilon \frac{\nabla L(W)}{\sqrt{r} + \delta}$
- Hyperparameter ρ, e.g. 0.9

Optimization methods - adam

- Adam = Adaptive moments,
  combines the ideas of momentum and scaling the gradients
- Momentum, $m = \beta_1 m + (1 - \beta_1)\nabla L(W)$
- Accumulate squared gradient, $r = \beta_2 r + (1 - \beta_2)\nabla L(W)^2$
- Update weights, $W = W - \epsilon \frac{m}{\sqrt{r}+\delta}$
- Default hyper parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ works in most cases
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Optimization methods - comparison



Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Optimization methods - comparison

- See visual comparison at https://ruder.io/optimizing-gradient-descent/
- SGD is slow, can get stuck at saddle points
- Momentum is fast but more unstable?
- AdaGrad and Rmsprop are "smarter" ?

Optimization methods - Keras

```
from keras.models import Sequential, Model
from keras.losses import binary_crossentropy as BC
from keras.optimizers import SGD, Adam

model1 = Sequential()
model2 = Sequential()
...
model1.compile(loss=BC, optimizer=SGD(lr=0.05),
               metrics=['accuracy'])

model2.compile(loss=BC, optimizer=Adam(lr=0.05),
               metrics=['accuracy'])
```

Optimization methods - Keras

- https://keras.io/api/optimizers/

  Available optimizers

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

- How to set the learning rate?
- Start with the default value, look at the training curve
- If loss / accuracy is very noisy, reduce learning rate
- If training requires several hundred epochs,
  try increasing the learning rate

Initialization of weights (8.4)

- Never initialize all weights to 0! Gives a linear model
- Before deep learning, the weights in a neural network were often initialized using random numbers from standard Gaussian (mean 0, variance 1)
- In deep learning, the initialization is more important due to many layers in a network (numerical stability) (and also depends on the activation function)
- Too large weights: exploding gradients
- Too small weights: vanishing gradients

Vanishing / exploding gradients

- For a network with 100 layers, a single weight w in each layer, no bias, linear activation functions, the output at the last layer will be $w^{100}x$
- $w^{100}x$ will be very large if $w > 1$
- $w^{100}x$ will be very small if $w < 1$

Initialization of weights

- Initalization adapted to layer with m inputs, n outputs

  $$W_{i,j} = U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

- Xavier (Glorot) initialization,

  $$W_{i,j} = U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

- He initialization,

  $$W_{i,j} = \sqrt{\frac{2}{m}} \cdot N(0,1)$$

- See https://keras.io/initializers/ for details

- **Dense** [source]

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform',
```

Initialization of weights

- Why divide random numbers by m or n ?
- A single node in a network will calculate
  $s = w_1 x_1 + w_2 x_2 + ... + w_m x_m + b$
- If the number of input dimensions m is large, s can also become large
- Dividing by m will make the gradients behave nicer

Batch normalization (8.7.1)

- A technique for improving speed, performance and stability when training deep neural networks
- Batch B with m training examples,
  calculate mean and variance (per dimension / unit)
- $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$, $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
- Normalize each dimension k separately
- $\hat{x}_i^k = \frac{x_i^k - \mu_B^k}{\sqrt{\sigma_B^{k2} + \delta}}$
- $\hat{x}^k$ now has zero mean and unit variance

Batch normalization

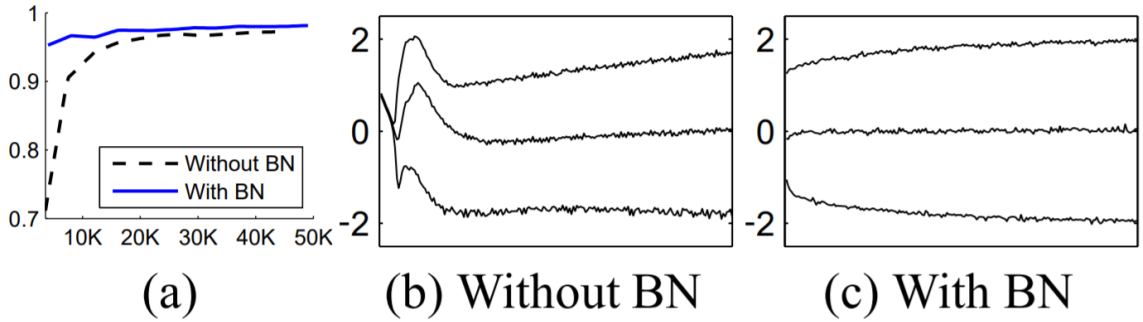


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Ioffe, Sergey; Szegedy, Christian (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". arXiv:1502.03167

Batch normalization

- Why is batch normalization important for deep networks?
- Compare with normalizing inputs x,
  remove mean and divide by standard deviation
- Normalizing the input makes optimization easier,
  since the loss function behaves nicer (more isotropic)
- Instead of only normalizing the inputs x,
  normalize the output of every layer

Other normalizations

- Instance normalization is another type of normalization
- Batch norm: calculate mean and std per dimension
- Instance norm: calculate mean and std per training example
- N = size of batch, C = channels, H,W = 1D representation of outputs in channel

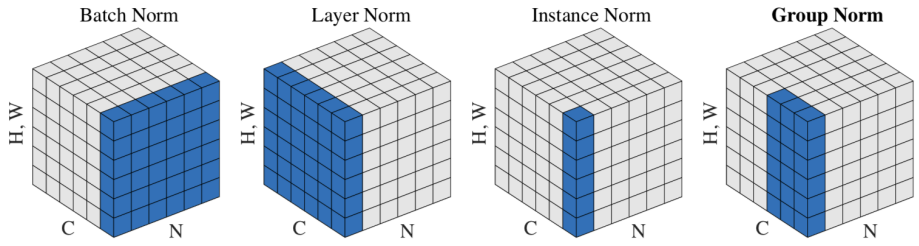


Figure 2. Normalization methods. Each subplot shows a feature map tensor. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Group Norm is illustrated using a group number of 2.

- Wu, Y., & He, K. (2018). Group normalization. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 3-19).

Batch normalization - Keras

```
from keras.layers.normalization import
BatchNormalization

model.add(Dense(10, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(10, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(10, activation='relu'))
model.add(BatchNormalization())
```

See https://keras.io/api/layers/normalization_layers/

keras-contrib provides instance normalization