MSE | MASTER OF SCIENCE IN ENGINEERING

(More) Efficient Learning of DNNs

TSM_DeLearn

Jean Hennebert
Martin Melchior

Overview

Recaps

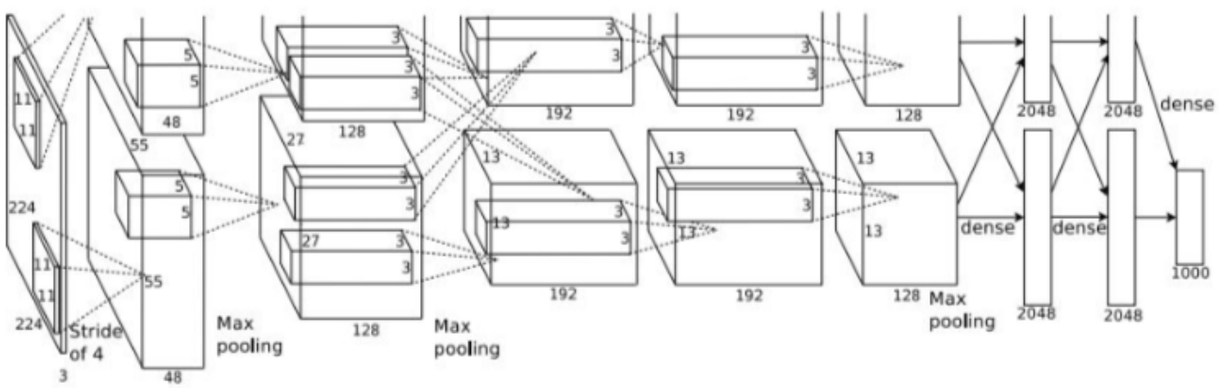
Input Normalisation

Activation Functions

Parameter Initialisation

Batch Normalisation

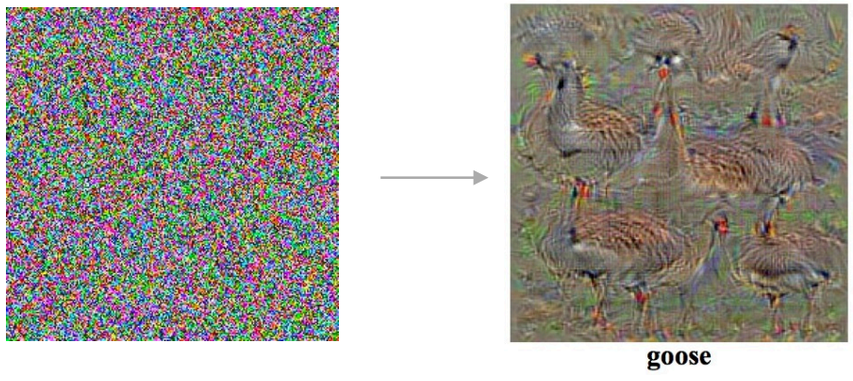
Recap 1: Hierarchical Features, Looking behind the Scene of CNNs



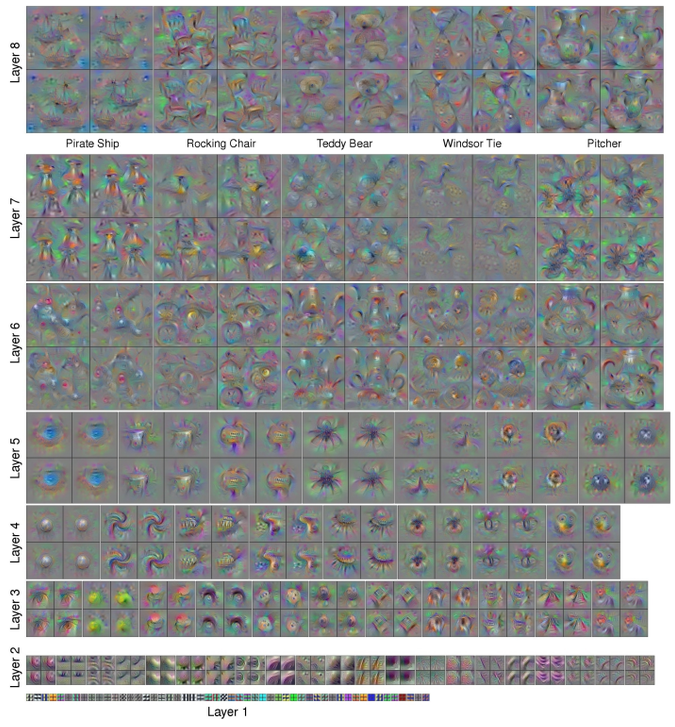
AlexNet
number of filters increasing with depth, more higher-level features

Visualisations

Inputs that maximise activations



yosinski.com/deepvis



Recap 2: Data Augmentation

Iff applicable

Rotate, Flip, Crop, Zoom

Shear, Elastic Deformations

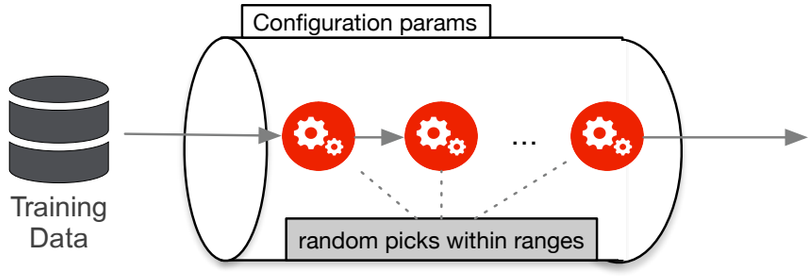
Transformations in color space

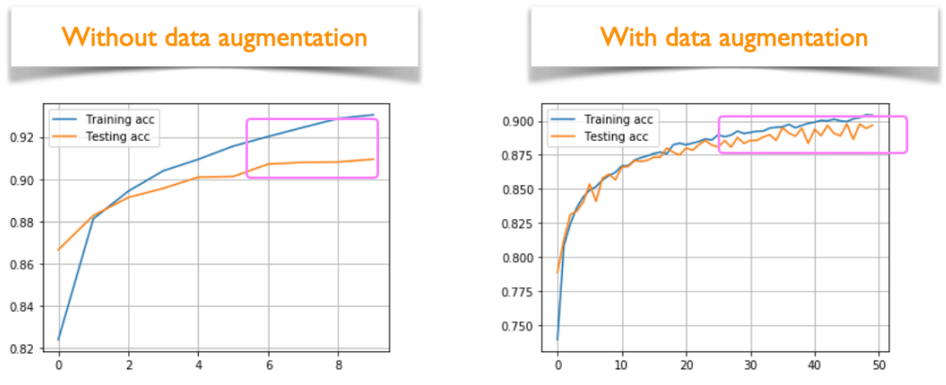
Regularisation Method

Generalises the model to also cope with transformed inputs.

Possibly, makes it “invariant” w.r.t. these transformations.

Pre- vs Online Augmentation

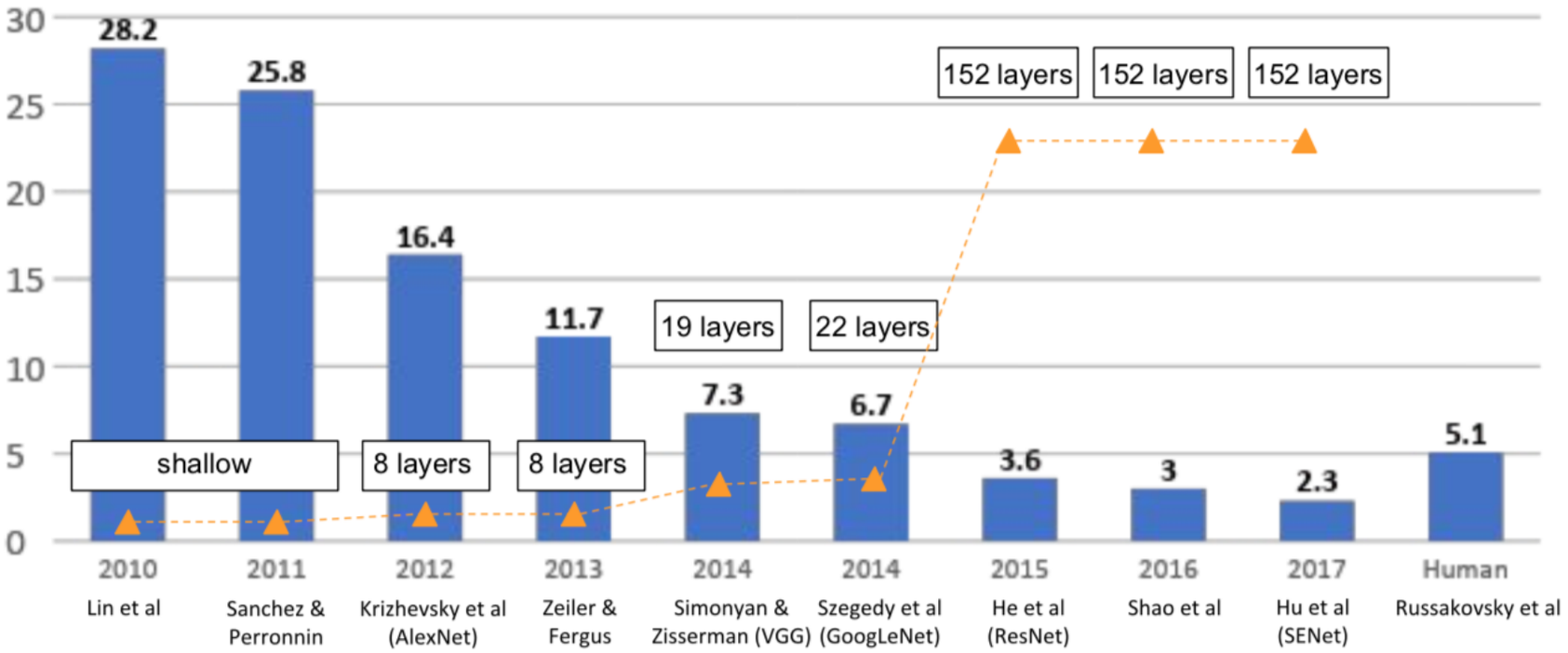




 For implementation in pytorch, see https://pytorch.org/vision/master/transforms.html

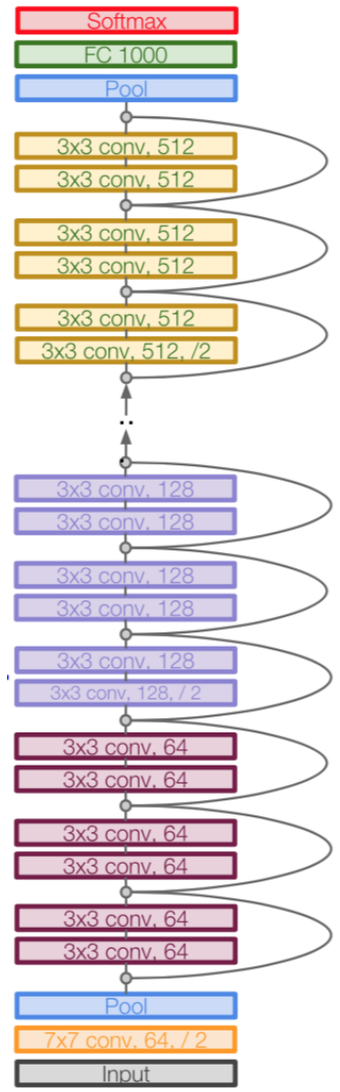
Recap 3: Deep CNN Architectures

ImageNet Competition



ResNet

with up to 152 layers



Deeper Architectures since 2012

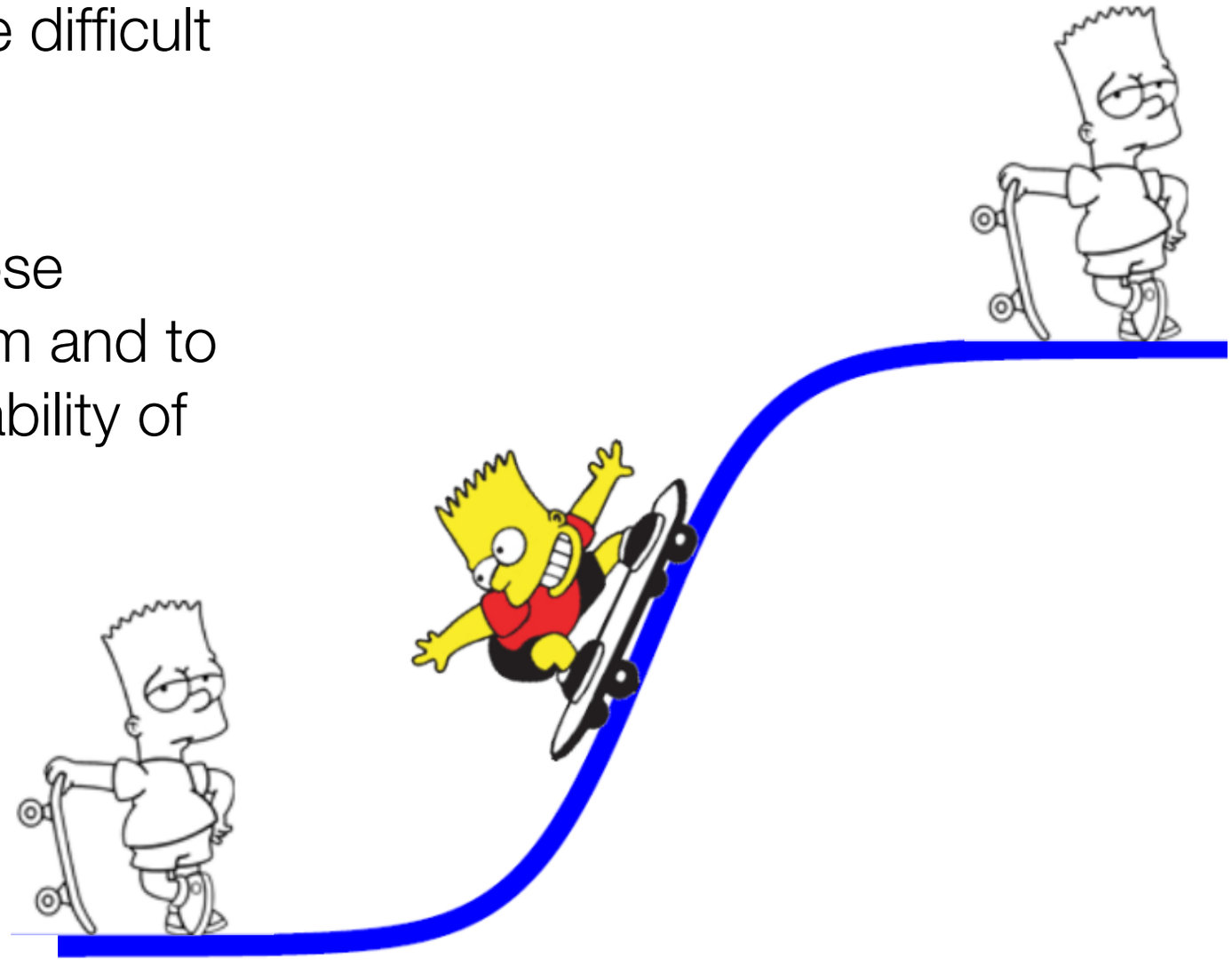
Specific improvements to get that deep:

Special architectural elements (e.g. skip connections, auxiliary outputs), batchnorm, weights initialisation, etc.

Goal of Todays Lecture

- Understand the vanishing gradient problem: Deeper learning models (increasing number of layers) become increasingly more difficult to train.

- Learn techniques to oppose vanishing gradient problem and to improve efficiency and stability of training:
  - Input Normalisation
  - Batch Normalisation
  - Parameter Initialisation
  - Activation Functions



Important Aspects to Highlight Today

- Distribution of values formed by the linear affine transformations (MLP, CNN)
- Try to match with the ranges of values the dynamic range of the activation functions,
- both in the forward and the backward path,
- over the succession of the layers.

$$x^{[l+1]} = \sigma\left(x^{[l]}\left(W^{[l]}\right)^{\intercal} + b^{[l]}\right)$$

Type of Activation Function

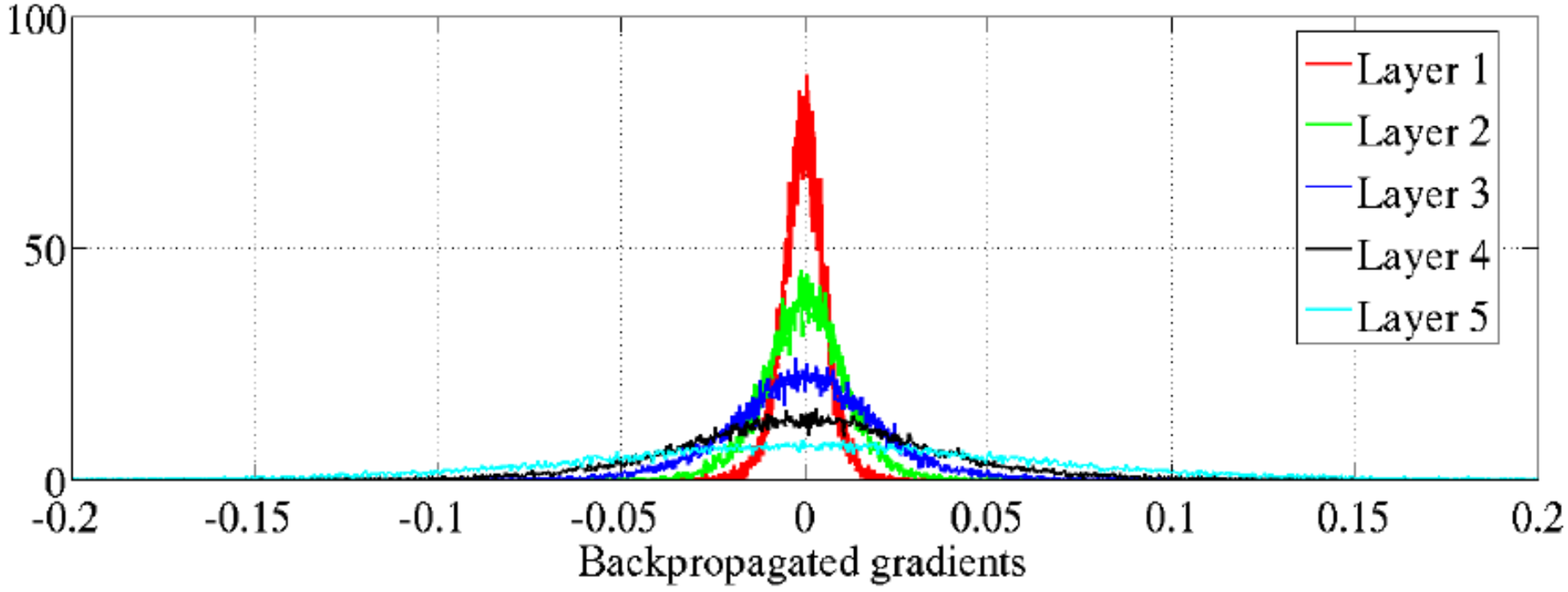
Normalisation
- input
- per batch

Initialisation of Parameters

for MLP and CNNs

Vanishing Gradients

- Gradient is computed by flowing back through network.
- Distribution of gradients in earlier layers smaller than in later layers.
- Why?



From a paper of X.Glorot & J.Bengio, 2010 (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf):

Overview

Recaps

Input Normalisation

Activation Functions

Batch Normalisation

Parameter Initialisation

Input Data Normalisation

- Make sure that the feature values in the input data are
  - On similar scales: *Feature scaling*.
  - Centred around zero: *Feature centering*.

- Why?
  - Make them independent of arbitrary physical units.
  - Learning can focus on identifying patterns in the distribution of the features without first correcting for scale and location.
  - It improves the *numerical stability* of the learning algorithm.
  - Improves the *convergence* of the learning algorithm.
  - Rules for *initialising the weights* can be formulated *independent* of the input data (see below).

z-Normalisation, Standardisation

$$x_k'^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma_k}$$

Centering and *rescaling* the data to obtain zero mean and unit-variance per feature (k).

where

$$\begin{aligned} \mu_k &= \frac{1}{N}\sum_i^N x_k^{(i)} \\ \sigma_k^2 &= \frac{1}{N}\sum_i^N (x_k^{(i)} - \mu_k)^2 \end{aligned}$$

$x_k^{(i)}$ — i: samples, k: features

Remarks:

- Parameters μ_k, σ_k must be *computed on the training set*.
- Normalisation of *image data*: Typically not per pixel; mean and standard deviation computed over all the pixels.

Min-Max Rescaling, Min-Max Normalisation

Min-Max Rescaling

$$x_k'^{(i)} = \frac{x_k^{(i)} - \min_k}{\max_k - \min_k}$$

Features are rescaled to [0,1] range.

Min-Max Normalisation

$$x_k'^{(i)} = 2 \cdot \frac{x_k^{(i)} - \min_k}{\max_k - \min_k} - 1$$

Features are rescaled and centred to [-1,1] range.

where $\min_k = \min_i \{x_k^{(i)}\}$, $\max_k = \max_i \{x_k^{(i)}\}$

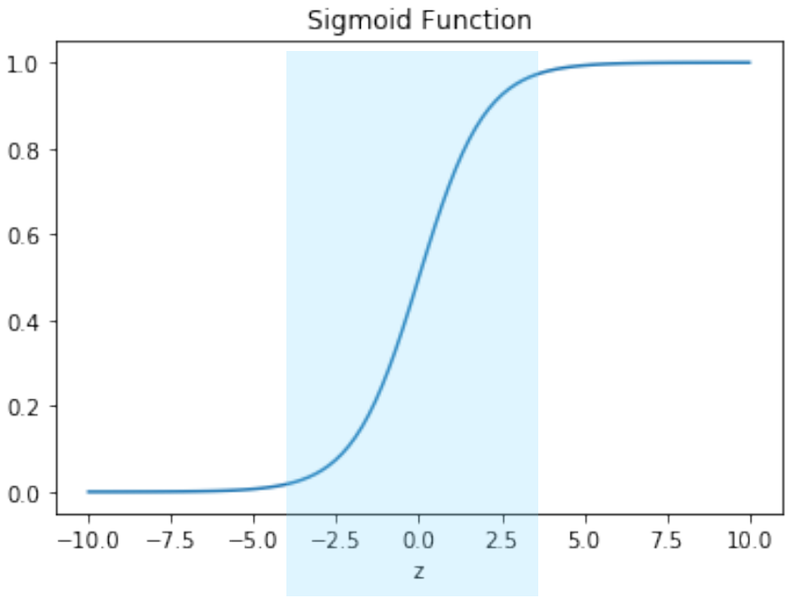
- min/max computed on the basis of training set
- image data: min/max computed over all pixels

Why is Feature Scaling Important?

Consider first layer of MLP with sigmoid activation function:

$$z = xW^{\intercal} + b\,, a = \sigma(z)$$

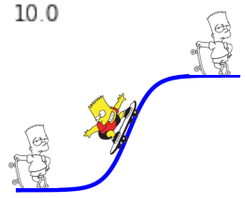
For efficient learning, we should reach with the bulk of the z-values the interesting dynamic range of the activation function.



interesting
dynamical range

Without Feature Scaling

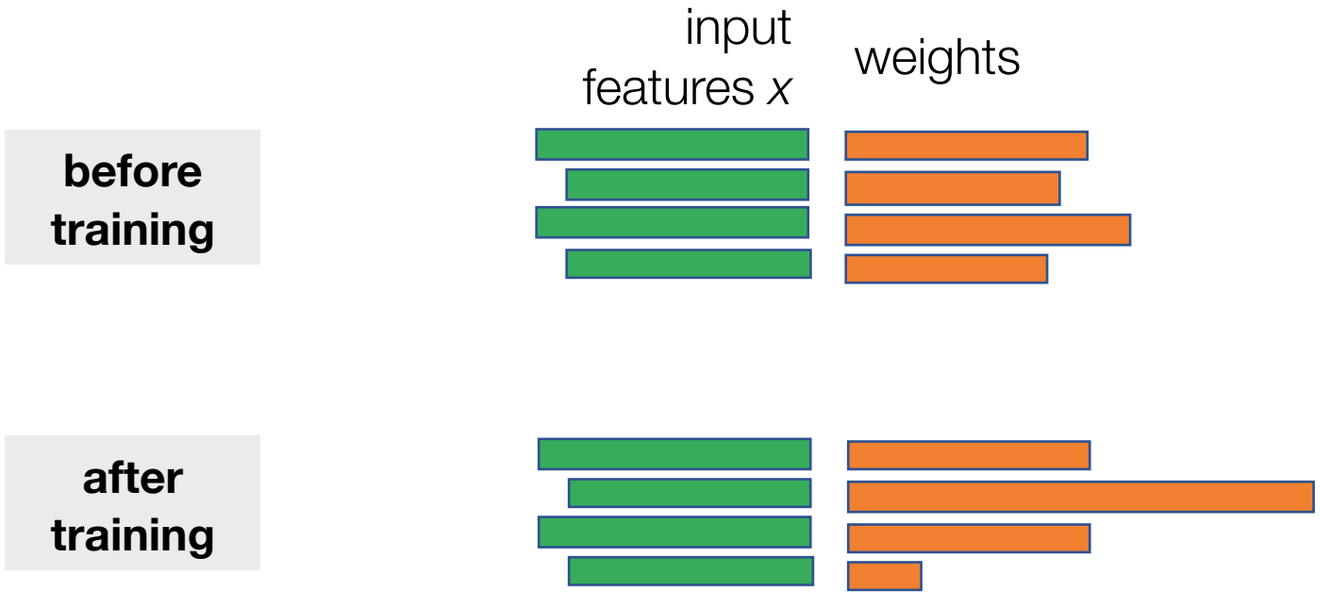
Either:
Initialisation of weights *dependent* of input data (and of specific model)
Or:
Features with large magnitude will dominate the variation of z and learning first needs to focus on handling the different scales (before handling their importance).



Why is Feature Scaling Important?

With Feature Scaling

Normalise the inputs to make them scale-independent. This allows for initialising the weights independently of the input data and learning can concentrate (from the beginning) on finding the weights relevant for the task.



Scale of the weights w can be initialised independently while still keeping z in a reasonable range.

Weights can directly be interpreted as feature importance!

Why is Feature Scaling Important?

From the chapter on backprop

$$\Delta W^{[1]} \propto \frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial z^{[1]}} \left(a^{[0]}\right)^{\intercal} \propto x^{\intercal}$$

For large scale features:

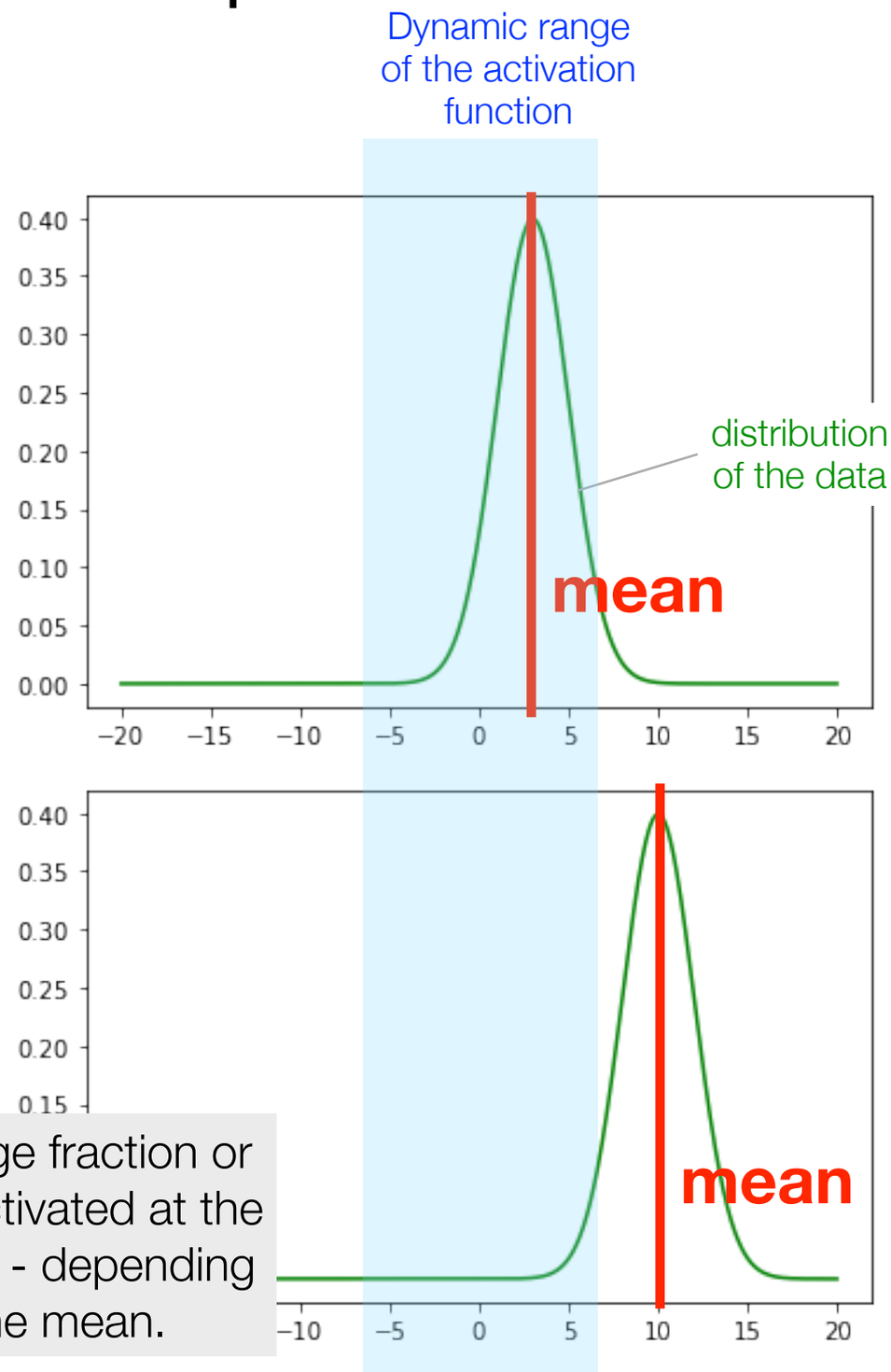
- we obtain large updates for the weights
- small weights, i.e. inversely proportional to feature scale, are needed
- which are hard to reach with large updates.

Why is Feature Centering be Important?

Similar argument as above: Bulk of the values need to be well within the dynamic range of the activation function.

With centralisation the training can directly focus on learning a bias from the neutral position where suffici

Without centralisation the training needs to first move the distribution into the dynamic range of the activation function which may take long.



Without bias term, a large fraction or all of the data may be activated at the beginning of the training - depending on the location of the mean.

Don’t Apply Normalisation Blindly

- Carefully evaluate the source data to be normalised.
  - Example: When sampling the data from a process with a trend, consider de-trending first.

- Standardisation or Min-Max Scaling? - No obvious answer, depends on data and task to be learned and model.
  - Use Standardisation if features have symmetric Gaussian-like distribution or if there are no natural lower and upper bounds.
  - Use Min-Max Scaling (with or without centering) when there are natural lower and/or upper bounds.

- Further References:
  - Watch what Hinton says: https://www.youtube.com/watch?v=Xjtu1L7RwVM&list=PLoRl3Ht4JOcdU872GhiYWf6jwrk_SNhz9&index=26
  - Blogs articles:
    https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35
    https://sebastianraschka.com/Articles/2014_about_feature_scaling.html

Input Normalisation in *PyTorch*

```
import numpy as np
import torch
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Normalize, Compose
```

```
training_data = datasets.mnist.FashionMNIST(root="data", train=True,
                                download=True, transform=ToTensor())
test_data = datasets.mnist.FashionMNIST(root="data", train=False,
                                download=True, transform=ToTensor())
```

```
loader = DataLoader(training_data, batch_size=60000, shuffle=False, num_workers=2)
X,_ = next(iter(loader))
mean, std = torch.mean(X).item(), torch.std(X).item()
print(mean,std)
```

```
0.2860405743122101 0.3530242443084717
```

determine mean and stdev for the training set

```
training_data = datasets.mnist.FashionMNIST(root="data", train=True,
            download=True, transform=Compose([ToTensor(), Normalize(mean,std)]))
test_data = datasets.mnist.FashionMNIST(root="data", train=False,
            download=True, transform=Compose([ToTensor(), Normalize(mean,std)]))
```

Use a composition of transforms to do the normalisation, use the mean and stdev of training set

```
loader = DataLoader(training_data, batch_size=1024, shuffle=False, num_workers=2)
X,_ = next(iter(loader))
print(torch.mean(X), torch.std(X), torch.min(X), torch.max(X))
```

```
tensor(-0.0075) tensor(1.0019) tensor(-0.8103) tensor(2.0224)
```

batches have roughly mean=0 and stdev=1

Input Normalisation in Keras

```
tf.keras.layers.Normalization(
    axis=-1, mean=None, variance=None, **kwargs
)
```

similar to data augmentation

```
inputs = keras.Input(shape=input_shape)
x = preprocessing_layer(inputs)
outputs = rest_of_the_model(x)
model = keras.Model(inputs, outputs)
```

Overview

Recaps

Input Normalisation

Activation Functions

Parameter Initialisation

Batch Normalisation

Role of Activation Function

- Single neuron in an MLP $h(x) = f(W \cdot x + b)$

  Affine transformation: $z = W \cdot x + b$ (linear + bias)

  Activation function: $a = f(z)$ (in most cases non-linear)

- Linear activation function

  If activation function is linear for all neurons, the mapping function for the neural network is linear and the representational capacity very limited.

  For a linear mapping, a network with a single layer is sufficient.

- Non-linearities crucial for representational capacity

  Non-linearities in the mapping between input and output of a neural network are crucial for gaining sufficient power for learning a task with sufficient accuracy.

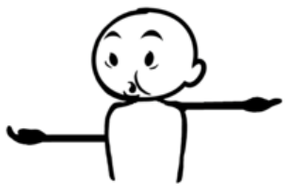
Sigmoid

$$y = \frac{1}{1+e^{-x}}$$

Tanh

$$y = \tanh(x)$$

Step Function



$$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$$

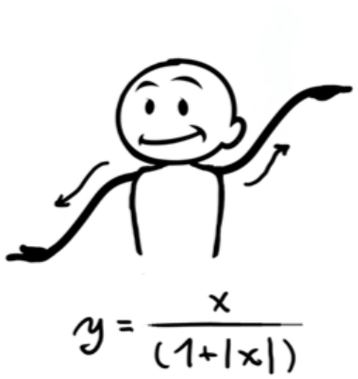
Softplus

$$y = \ln(1+e^{x})$$

ReLU

$$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$$

Softsign

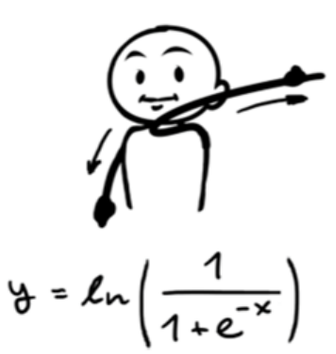


$$y = \frac{x}{(1+|x|)}$$

ELU

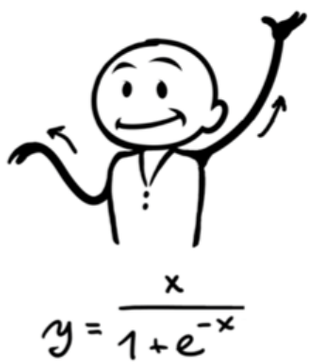
$$y = \begin{cases} \alpha(e^{x}-1), & x<0 \\ x, & x \geq 0 \end{cases}$$

Log of Sigmoid



$$y = \ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish



$$y = \frac{x}{1+e^{-x}}$$

Sinc

$$y = \frac{\sin(x)}{x}$$

Leaky ReLU

$$y = \max(0.1x, x)$$

Mish

$$y = x(\tanh(\text{softplus}(x)))$$

from https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

Which Activation Function to Choose When?

- Representational capacity

  Non-linear - except for last layer in regression task

- Task specific / sometimes conventionally

  Binary classification (last layer)

  Multiclass classification (last layer)

  Regression task (last layer)

  Sequential

- Robustness and performance of learning algorithm

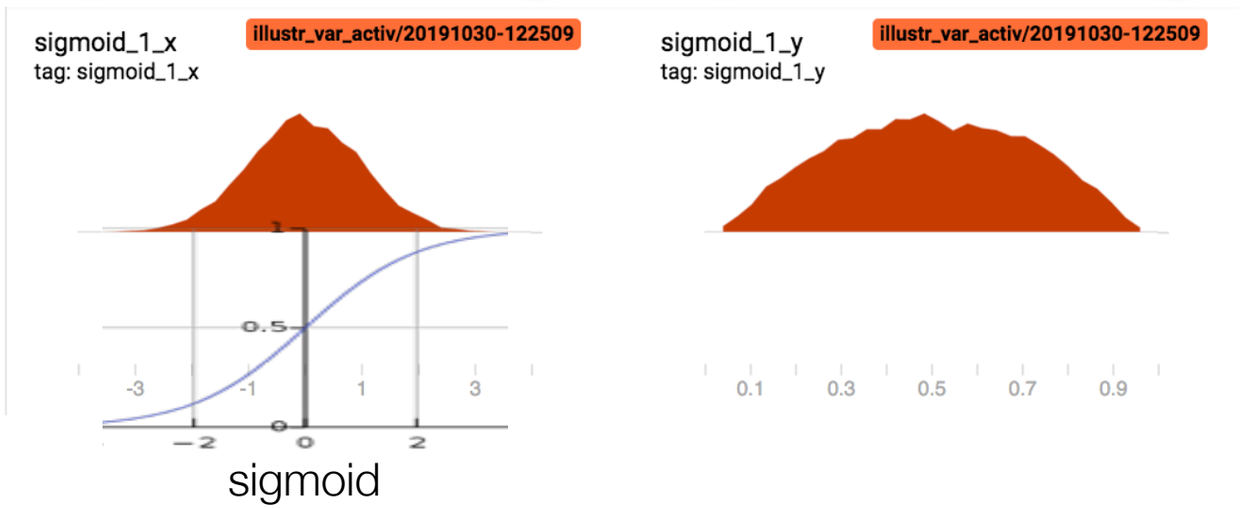
  See vanishing / exploding gradient problem in week 6

Saturation Regions in Sigmoid Function

Input x:
Gaussian Random Numbers

Output y:
passed through sigmoid function

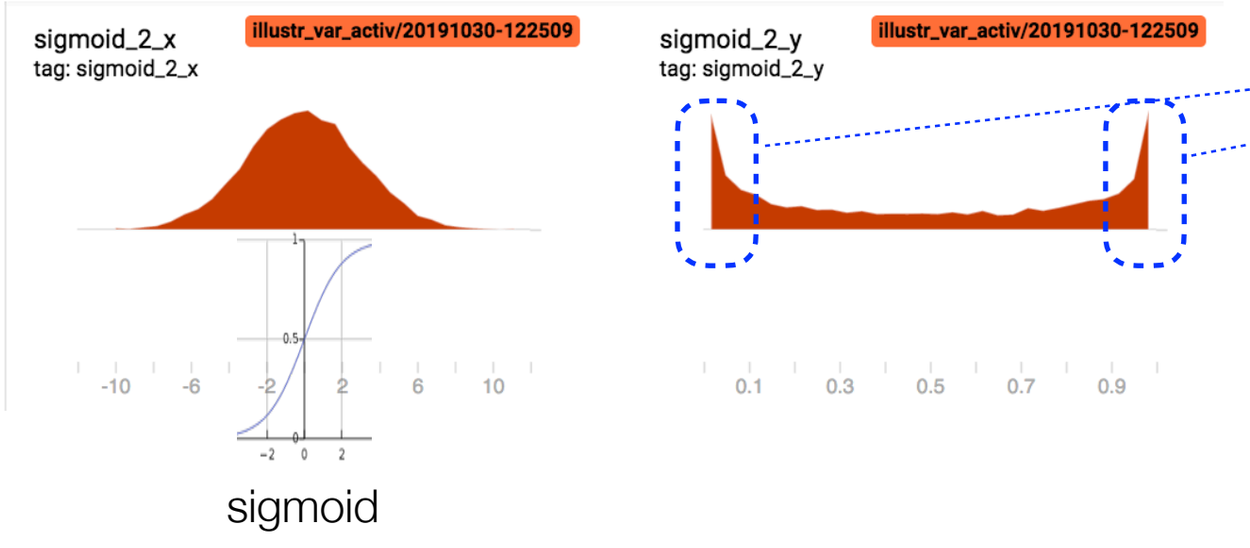
stdev (input):
1.0



sigmoid

Saturation region reached with low probability

stdev (input):
3.0



sigmoid

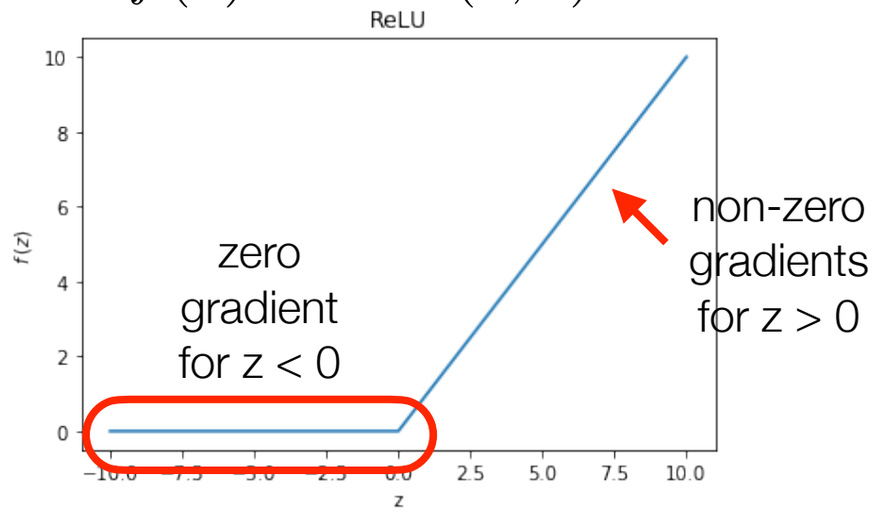
Saturation region reached with high probability

Gradients can become very small with high probability

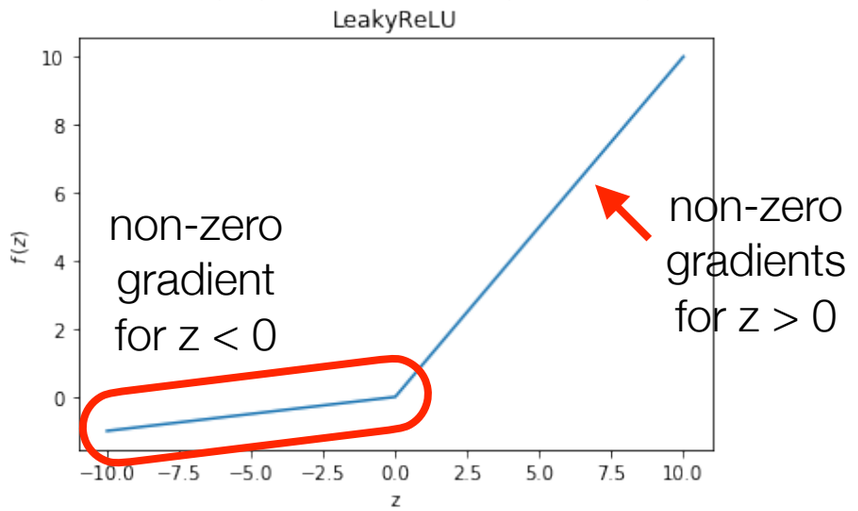
Non-Saturating Activation Functions

Avoid S-shaped activation functions that flatten out at larger z-magnitudes —> ReLU, LeakyReLU or ELU.

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



$$f(z) = \max(\eta z, z)$$



Dying units problem with ReLU

For negative logit, activations is 0 ***and*** derivative is 0 —> No weight update for these neurons and these neurons are likely to stay 'dead'.

Solution Leaky ReLU (or its smooth version, ELU).

The parameter η can be determined by hyper-parameter tuning or by setting a good default value ($\eta = 0.01$). See (*) for an evaluation of the different ReLUs.

(*) "Empirical Evaluation of Rectified Activations in Convolution Network," B. Xu et al. (2015).

Activation Functions in TF / Keras

Example: Model in line with the universal representation theorem

```
model = tf.keras.models.Sequential(
            [tf.keras.layers.Dense(100, activation='sigmoid', name="hidden1"),
             tf.keras.layers.Dense(1, activation='linear', name="output")]
        )
```

Choices: 'relu', 'tanh', 'selu

Possibly split definition of affine part and non-linear part (e.g. to squeeze in a batch norm layer)

Example: Some other non-sense model

```
model = tf.keras.models.Sequential(
            [tf.keras.layers.Dense(100, name="linear1"),
             tf.keras.layers.Activation(activation='sigmoid', name="non-linear1"),
             tf.keras.layers.Dense(100, name="linear2"),
             tf.keras.layers.ELU(alpha=1.5, name="elu")]
        )
```

Example for using an activation function with a parameter.

Overview

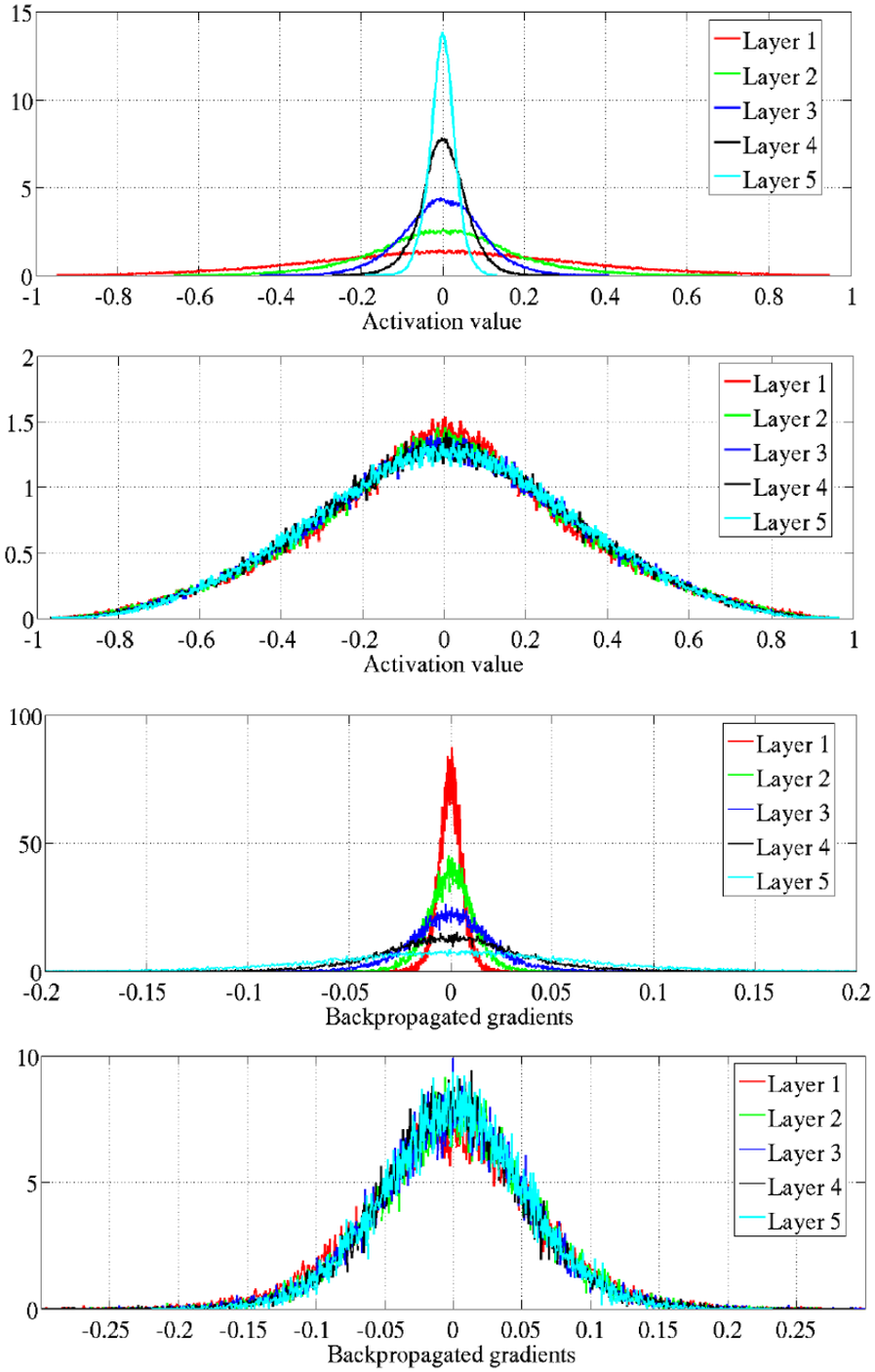
Recaps

Input Normalisation

Activation Functions

Parameter Initialisation

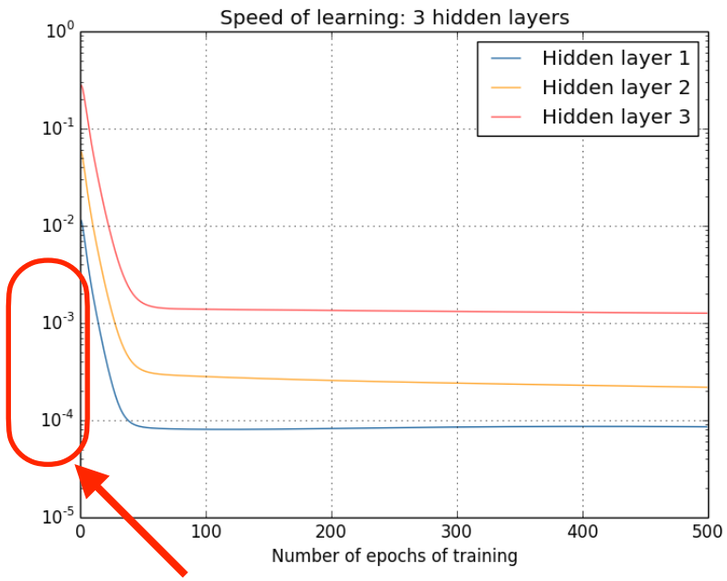
Batch Normalisation

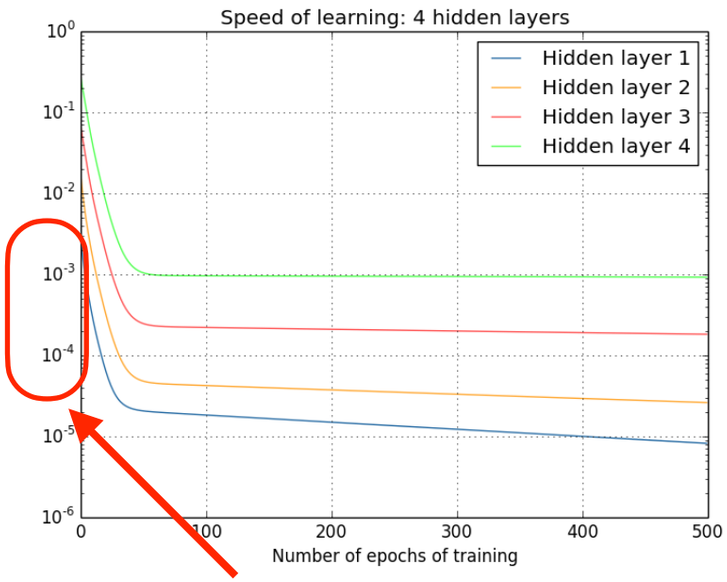


Vanishing Gradients at Early Layers

Learning MNIST with backprop, *standard normally distributed initial weights*, sigmoid activations (see neuralnetworksanddeeplearning.com):

Learning slow in earlier layers, slower the more layers there are.





Note that here, small gradients don't imply that we are close to the optimum!

Learning speed (~length of gradients) in the first two layers drops by an order of magnitude when using four instead of three hidden layers (see the different scales in the two figures)!

In other cases (e.g. RNNs) the opposite is observed, i.e. exploding gradients.

Complex Dynamics when Training DNNs

- Features in the input data need to be extracted and mapped by the DNN to the label information (in supervised learning).
- Information needs to flow forth (input signal) and back (correction signal) through the DNN over the course of the training.
- Slow learning at early layers if gradients get small. At beginning of training, features extracted by earlier layers are random, contain very little information. Parameters in later layers ignore this information. Not in line with hierarchical features.

Some Hints on Vanishing Gradients

From backprop:

$$\frac{\partial L}{\partial a^{[l-1]}} = \frac{\partial L}{\partial a^{[l]}} * \boxed{\sigma'(z^{[l]})} \cdot \boxed{W^{[l]}}$$

can be small in saturation regime of activation function

values should not become too small (vanishing gradients) or too large (exploding gradients.

Gradient vanishes exponentially with the depth if the weights are ill-conditioned or the logits are in the saturating domain of the activation functions.

Possible strategies to counter vanishing gradients:

- Design of ***weight initialisation*** aims at controlling the ***variance of the activations and the gradients*** such that weights evolve at the same rate across layers during training, and no layer reaches the saturation regime before others.
- Use ***non-saturating activation functions***.

Analysis of the Gradients

Paper of Glorot and Bengio, 2010 (!)
http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

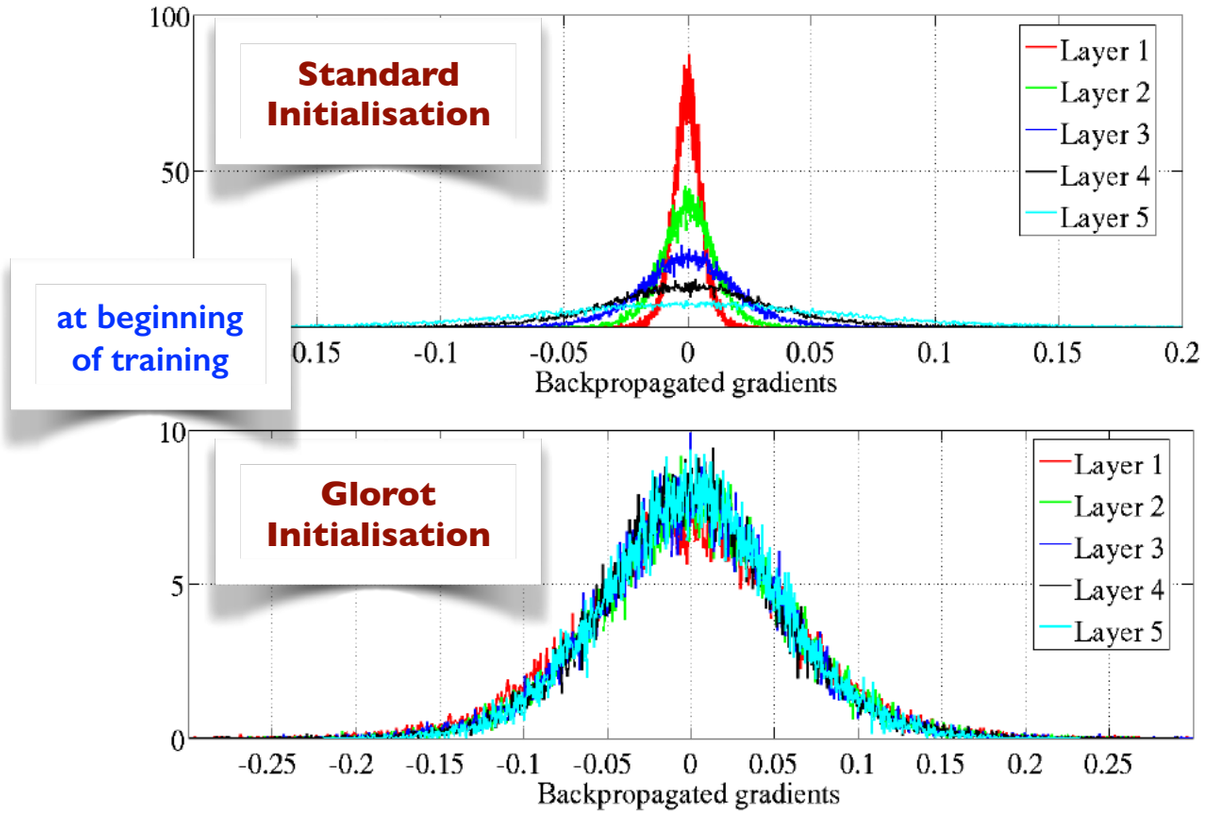


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Standard initialisation of weights at 2010 (for all layers weights $W^{[l]} \sim \mathcal{N}(0,1)$:
Magnitude of gradient components smaller for earlier layers.

Glorot & Bengio could design an initialisation strategy so that the gradient components have similar magnitudes across all layers.

Details: Stabilising Forward Pass

Assume we are in the dynamic range of the activation function, i.e. $\sigma(z) \approx c\, z$ where c is a constant:

$$a_j^{[l]} = \sigma\left(\sum_k^{N_{l-1}} W_{jk}^{[l]}\, a_k^{[l-1]}\right) \approx c \sum_k^{N_{l-1}} W_{jk}^{[l]}\, a_k^{[l-1]}$$

Assuming that all the weights in a layer are independent, identically distributed and independent from the activations:

$$\begin{aligned} \mathrm{var}\left[a^{[l]}\right] &\approx c\, N_{l-1} \mathrm{var}\left[W^{[l]}\right] \mathrm{var}\left[a^{[l-1]}\right] \\ &= \mathrm{var}\left[a^{[0]}\right] \prod_{j=1}^{l} c\, N_{j-1} \mathrm{var}\left[W^{[j]}\right] \end{aligned}$$

ignoring indices

use the following identifies for X,Y independent:

$Var(X+Y) = Var(X) + Var(Y)$

$Var(XY) = Var(X)Var(Y) + (E[X])^2 Var(Y) + Var(X)(E[Y])^2$

Choose:

$$\mathrm{var}\left[W^{[j]}\right] = \frac{1}{cN_{j-1}}$$

“gain” “fan-in”

Details: Stabilising Backward Pass

Similarly, with $\sigma'(z) \approx c$:

$$\frac{\partial L}{\partial a_j^{[l-1]}} = \sum_{k=1}^{N_l} \frac{\partial L}{\partial a_k^{[l]}} \sigma'(z_j^{[l]}) W_{kj}^{[l]} \approx c \sum_{k=1}^{N_l} \frac{\partial L}{\partial a_k^{[l]}} W_{kj}^{[l]}$$

and

$$\begin{aligned} \mathrm{var}\left[\frac{\partial L}{\partial a^{[l-1]}}\right] &\approx c N_l \mathrm{var}\left[\frac{\partial L}{\partial a^{[l]}}\right] \mathrm{var}\left[W^{[l]}\right] \\ &= \mathrm{var}\left[\frac{\partial L}{\partial a^{[L]}}\right] \prod_{j=l}^{L} c N_j \mathrm{var}\left[W^{[j]}\right] \end{aligned}$$

Choose: $\mathrm{var}\left[W^{[j]}\right] = \frac{1}{cN_j}$ $\Rightarrow$ $\mathrm{var}\left[W^{[j]}\right] = \frac{2/c}{N_{j-1} + N_j}$

"fan-out"

Compromise of forward and backward pass

Parameter Initialisation

- ***Randomly initialise weights:***
  - Break symmetries at start of learning.

- ***Initialise weights at proper scale:***
  - Properly scaling the width of the distribution (depends on the random distribution used to generate the weights and the activation function):

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic (sigmoid)	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Xavier Glorot

Kaiming He

number of input and output connections **for a given layer**

Remark: Schemes do not guarantee to keep the input/output variance identical in both directions, but provide a good compromise and work well in practice.

Parameter Initialisation PyTorch <-> TF

Different defaults for PyTorch and TF (dense and conv2d layers):

PyTorch	Weights	$\mathcal{U}(-s, s), s = \sqrt{1/N_{l-1}}$	Kaiming He (leaky ReLU)
	Bias	$\mathcal{U}(-s, s), s = 1/\sqrt{N_{l-1}}$	
Tensorflow	Weights	$\mathcal{U}(-s, s), s = \sqrt{6/(N_{l-1} + N_l)}$	Xavier Glorot (Tanh)
	Bias	0	

Parameter Initialisation PyTorch

Use `torch.nn.init`:

- `xavier_normal_(w)`, `xavier_uniform_(w)`
- `kaiming_normal_(w)`, `kaiming_uniform_(w)`
- `normal_(w)`, `uniform_(w)`

underscore at the end of method name for functions that in-place replace input tensor

```python
import math
from torch import Tensor
from torch.nn.init import _calculate_fan_in_and_fan_out, _no_grad_uniform_

def xavier_uniform_(tensor: Tensor, gain: float = 1.) -> Tensor:
    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / float(fan_in + fan_out))
    a = math.sqrt(3.0) * std  # Calculate uniform bounds from standard deviation
    return _no_grad_uniform_(tensor, -a, a)
```

from pytorch source code

compute “units” of given and previous layer

compromise of forward and backward pass

bounds specific for a uniform random variable with given standard deviation

Parameter Initialisation in PyTorch

```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = torch.nn.Flatten()
        self.linear1 = torch.nn.Linear(3*32*32, 600)
        self.activ1 = torch.nn.Sigmoid()
        self.linear2 = torch.nn.Linear(600, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.activ1(x)
        return self.linear2(x)

model = MLP()

w2 = model.linear2.weight
print(torch.std(w2.view(-1)).item())
torch.nn.init.kaiming_uniform_(w2, mode='fan_in', a=math.sqrt(5))
print(torch.std(w2.view(-1)).item())

torch.nn.init.xavier_normal_(w2)
print(torch.std(w2.view(-1)).item())
```

```
0.0237202420830726 62
0.0234924219548702 24
0.0581413134932518
```

By default initialisation scheme of Kaiming He with leaky ReLu and Uniform distribution;
for both, linear and CNN layers.

Customisation of parameter initialisation is easy; typically at model instantiation time.

Parameter Initialisation TF/Keras

Use `tf.keras.initializers`:

- `Glorot_Normal()`, `Glorot_Uniform()`
- `He_Normal()`, `He_Uniform()`
- `Random_Normal()`, `Random_Uniform()`

Instead of first names, given names are used for Xavier Glorot or Kaiming He, respectively.

Pass to the constructor of the layer an instance of the initializer class for the kernel_initializer argument

```
Dense(nunits, kernel_initializer=Glorot_Uniform(), activation='tanh')
```

Parameter Initialisation in TF / Keras

See initializers available in tensorflow/keras at
https://www.tensorflow.org/api_docs/python/tf/initializers

```
from tensorflow.keras import initializers
initializer = initializers.GlorotNormal()

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(100, activation='relu',
                                kernel_initializer=initializer))
model.add(tf.keras.layers.Dense(10, activation='softmax',
                                kernel_initializer=initializer))
```

Import initialiser and
Instantiate initialiser object

pass initializer object to constructor of layer:

- **`kernel_initializer`** (for weights)
- **`bias_initialiser`** (for bias)

or a string-identifier for an initializer-method

Use `tf.keras.initializers`:

- `Glorot_Normal()`, `Glorot_Uniform()`
- `He_Normal()`, `He_Uniform()`
- `Random_Normal()`, `Random_Uniform()`

In Tensorflow, the given names are used of Xavier Glorot or Kaiming He for identifying their initialisation methods - in contrast to PyTorch where first names are used..

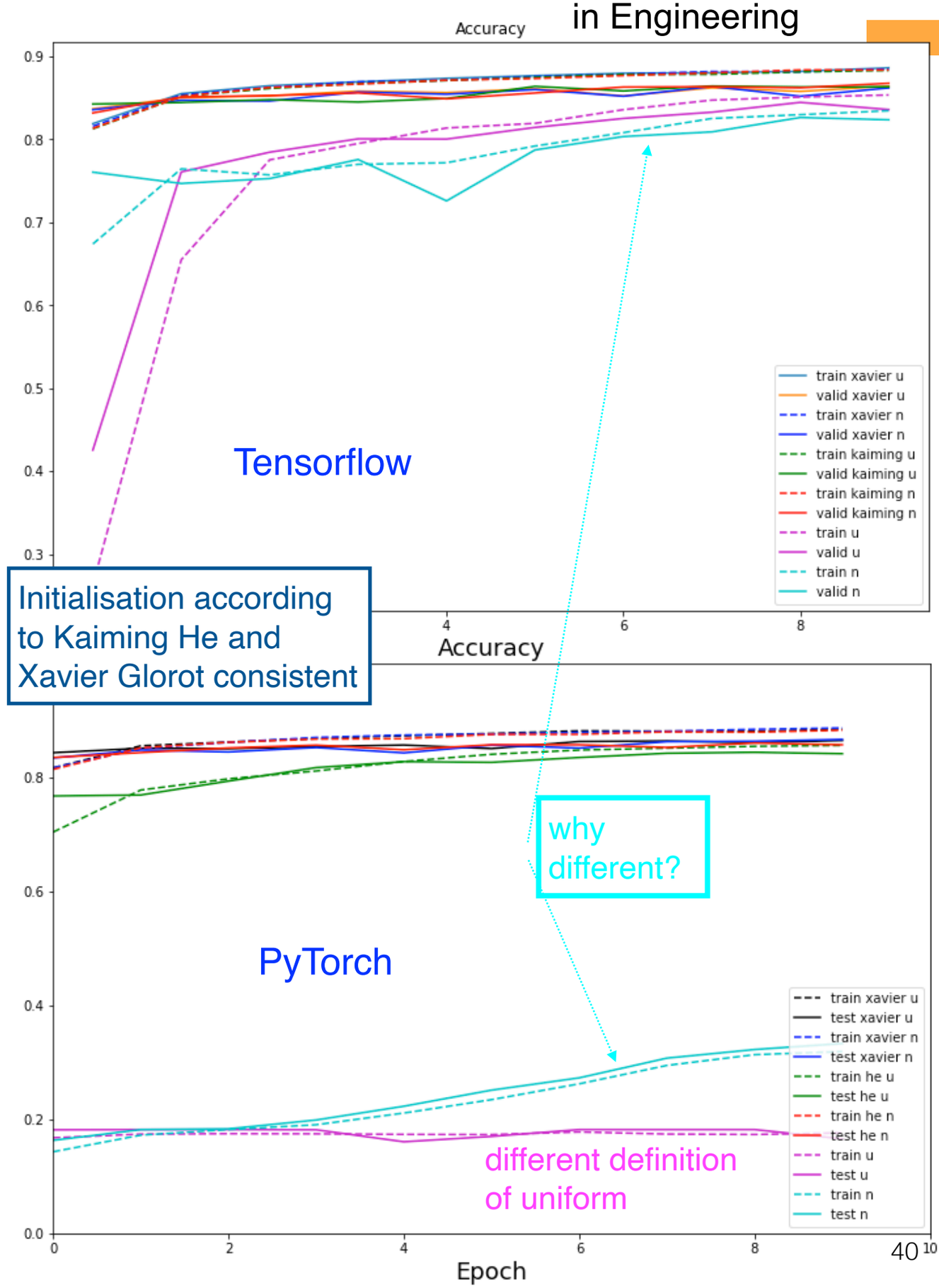
Example: Parameter Initialisation

MLP with

11 hidden layers (100 units, tanh), output layer with softmax.

Trained for Fashion-MNIST.

Adam with learning rate 0.001, batch size 64.



Overview

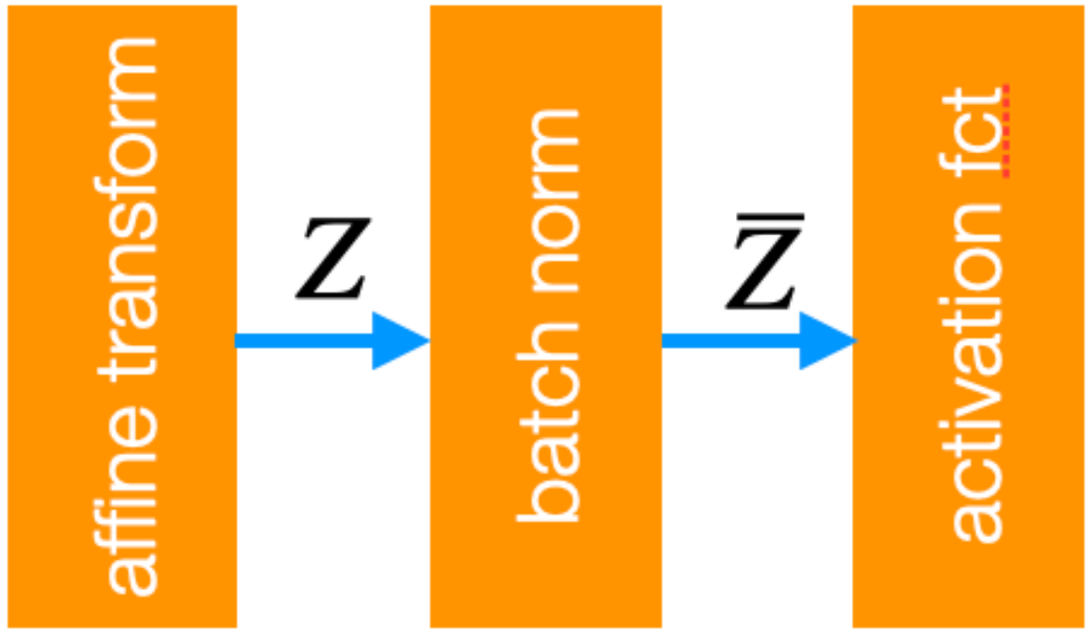
Recaps

Input Normalisation

Activation Functions

Parameter Initialisation

Batch Normalisation

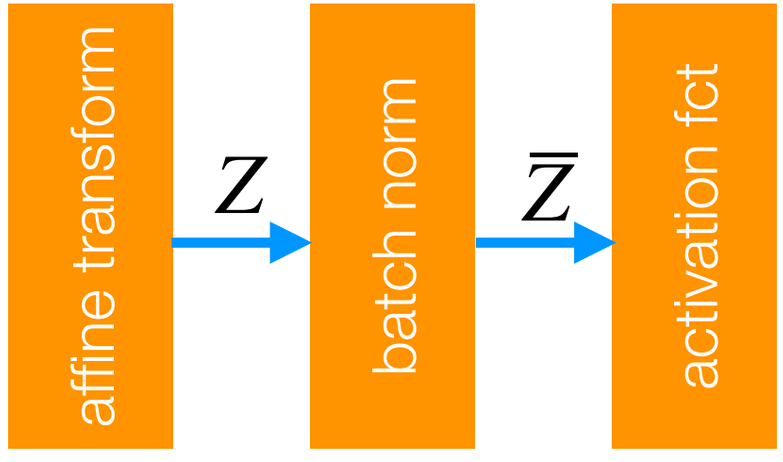


Batch Normalisation

Idea introduced by S. Ioffe and C. Szegedy(1):

- Normalise the logits Z of a layer with an estimate of the mean and stdev of the current mini-batch.
- Adjust the normalised logits $\overline{Z}$ with a trainable scale and shift parameter (γ, β).
- Apply the activation function.

Applicable to any layer in a network.



$$
\begin{aligned}
Z &= W \cdot A^{prev} + b \\
\tilde{Z} &= Z - \frac{1}{b}\sum_j Z^{(j)} \\
\hat{Z} &= \frac{\tilde{Z}}{\sqrt{\epsilon + \frac{1}{b}\sum_j \tilde{Z}^2}} \\
\overline{Z} &= \gamma \hat{Z} + \beta
\end{aligned}
$$

bias term can be ignored since it can be compensated with β

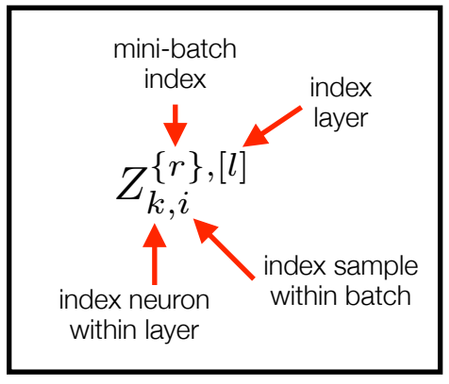
(1)S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” (2015).

Batch Norm Equations (training time)

In our notation:

Normalisation per mini-batch:

$$\left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} = \frac{Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}}{\sigma_k^{\{r\},[l]} + \epsilon}$$



where $\mu_k^{\{r\},[l]}$ and $\sigma_k^{\{r\},[l]}$ are given by

$$\begin{aligned}
\mu_k^{\{r\},[l]} &= \frac{1}{N_B}\sum_{i=1}^{N_B} Z_{k,i}^{\{r\},[l]} \\
(\sigma_k^{\{r\},[l]})^2 &= \frac{1}{N_B}\sum_{i=1}^{N_B}\left(Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}\right)^2
\end{aligned}$$

Scaling and **Shifting**:

$$\hat{Z}_{k,i}^{\{r\},[l]} = \gamma_k^{[l]}\left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} + \beta_k^{[l]}$$

where $\gamma_k^{[l]}$ and $\beta_k^{[l]}$ are parameters to be learned (i.e. optimised)

Batch norm changes backprop equations: Back prop through the normalisation (μ, σ) and consider derivates w.r.t. to the scale and shift parameters (γ, β).

Batch Norm Equations (production time)

At test time or in production we don't have batches!

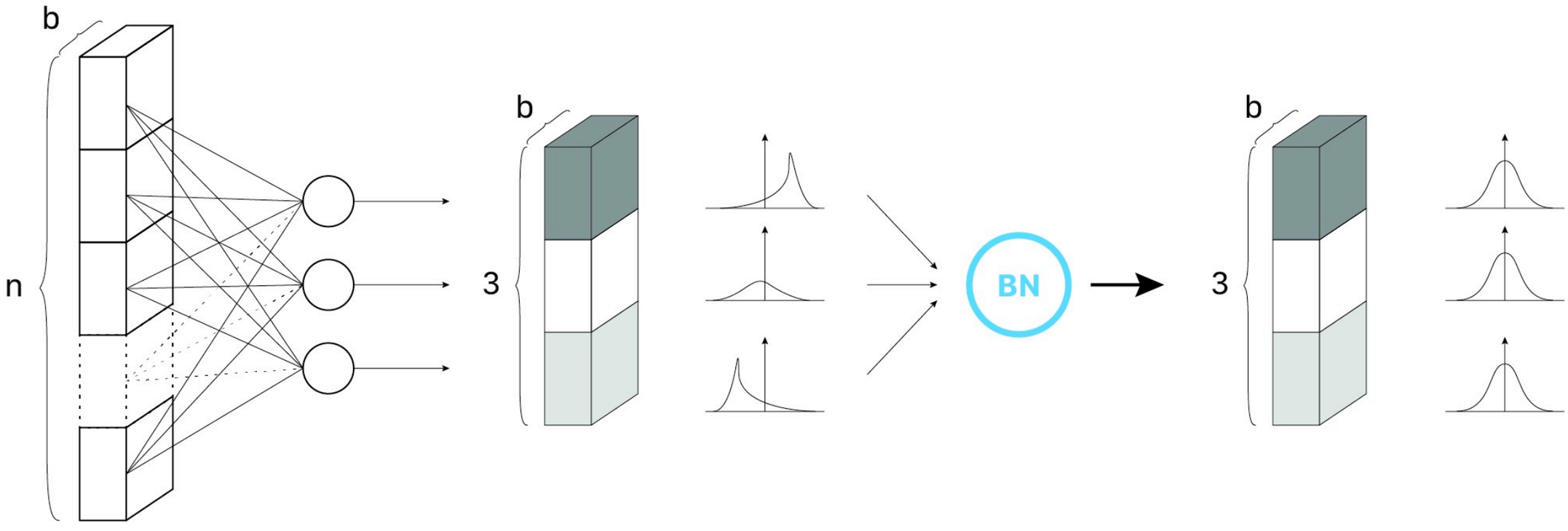
- The mean $\mu_k^{\{r\},[l]}$ and stdev $\sigma_k^{\{r\},[l]}$ computed per mini-batch during training are used to compute suitable mean and stdev for the population.
- These can be used at test time or for production. Typically, an exponentially weighted average is used — for a parameter ρ the update rule is defined through a running average $\hat{\rho}^{(r)}$ and decay rate ν by

$$\hat{\rho}^{(r+1)} = (1-\nu)\hat{\rho}^{(r)} + \nu\rho$$

- One such equation for each unit of a layer.

Effect of Batch Norm

Makes distribution of input values to each unit in a neural network distributed similarly to a standard normal distribution (during training) - before adjusting it with the learnable scale and shift parameter.



Network layer with 3 units, batch size b.

(Illustration from https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338)

Effect of Batch Norm

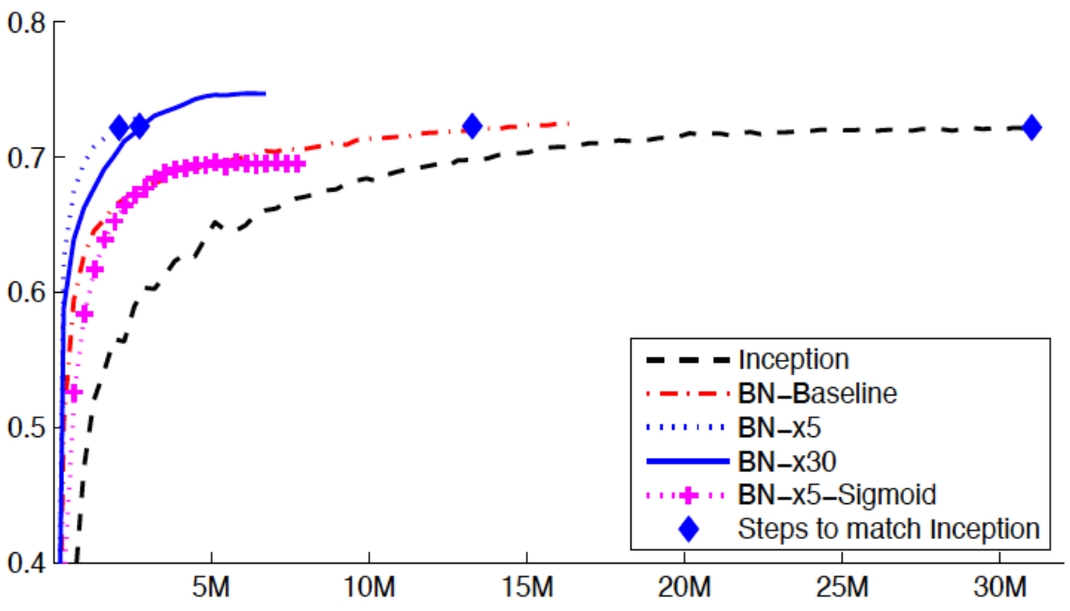


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

First applied to Inception Network (GoogleLeNet), taken as “baseline” (black line).

Have observed great speed-up in the training:

- Compared time to reach accuracy of baseline (steps to 72.2% in the table above)
- Could also increase learning rate (blue lines)
- Could even replace ReLU by Sigmoid.

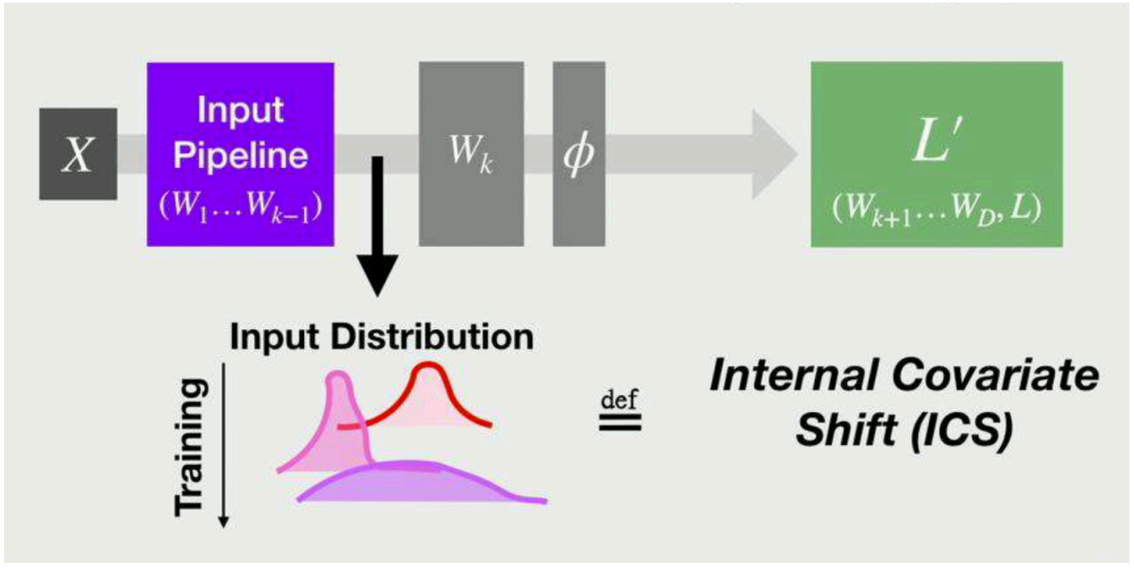
Effect of Batch Norm

Citing Goodfellow (see https://www.youtube.com/watch?v=Xogn6veSyxA):

Before BN, we thought that it was almost impossible to efficiently train deep models using sigmoid in the hidden layers. We considered several approaches to tackle training instability, such as looking for better initialization methods. Those pieces of solution were heavily heuristic, and way too fragile to be satisfactory. Batch Normalization makes those unstable networks trainable ; that's what this example shows.

Why does Batch Norm Work?

- Originally, Ioffe and Szegedy thought that it alleviates the internal covariate shift problem.



A given layer sees during training varying distributions of input values from previous layer - since the weights in the previous layers get updated and affect their activations. Layers need to continually adapt.

- S.Santukar et al (NIPS 2018) demonstrate that this hypothesis is wrong.
- Rather they hypothesise that
  - Batch Norm makes the updates in the different layers more independent.
  - Batch Norm makes the cost surface smoother (gradients change less rapidly), hence makes training more robust.

Batch Norm in PyTorch

```
class CNNBN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(3, 24, 5, bias=False)
        self.bn1 = torch.nn.BatchNorm2d(24)
        self.relu1 = torch.nn.ReLU()
        self.pool1 = torch.nn.MaxPool2d(2, 2)
        self.conv2 = torch.nn.Conv2d(24, 48, 5, bias=False)
        self.bn2 = torch.nn.BatchNorm2d(48)
        self.relu2 = torch.nn.ReLU()
        self.pool2 = torch.nn.MaxPool2d(2, 2)
        self.fc1 = torch.nn.Linear(48*5*5, 400, bias=False)
        self.bn3 = torch.nn.BatchNorm1d(400)
        self.relu3 = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(400, 80, bias=False)
        self.bn4 = torch.nn.BatchNorm1d(80)
        self.relu4 = torch.nn.ReLU()
        self.fc3 = torch.nn.Linear(80, 10)

    def forward(self, x):
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.relu3(self.bn3(self.fc1(x)))
        x = self.relu4(self.bn4(self.fc2(x)))
        x = self.fc3(x)
        return x
```

BatchNorm2d for 2d Convolutional Layers, i.e. for input tensors of shape (N,C,H,W)

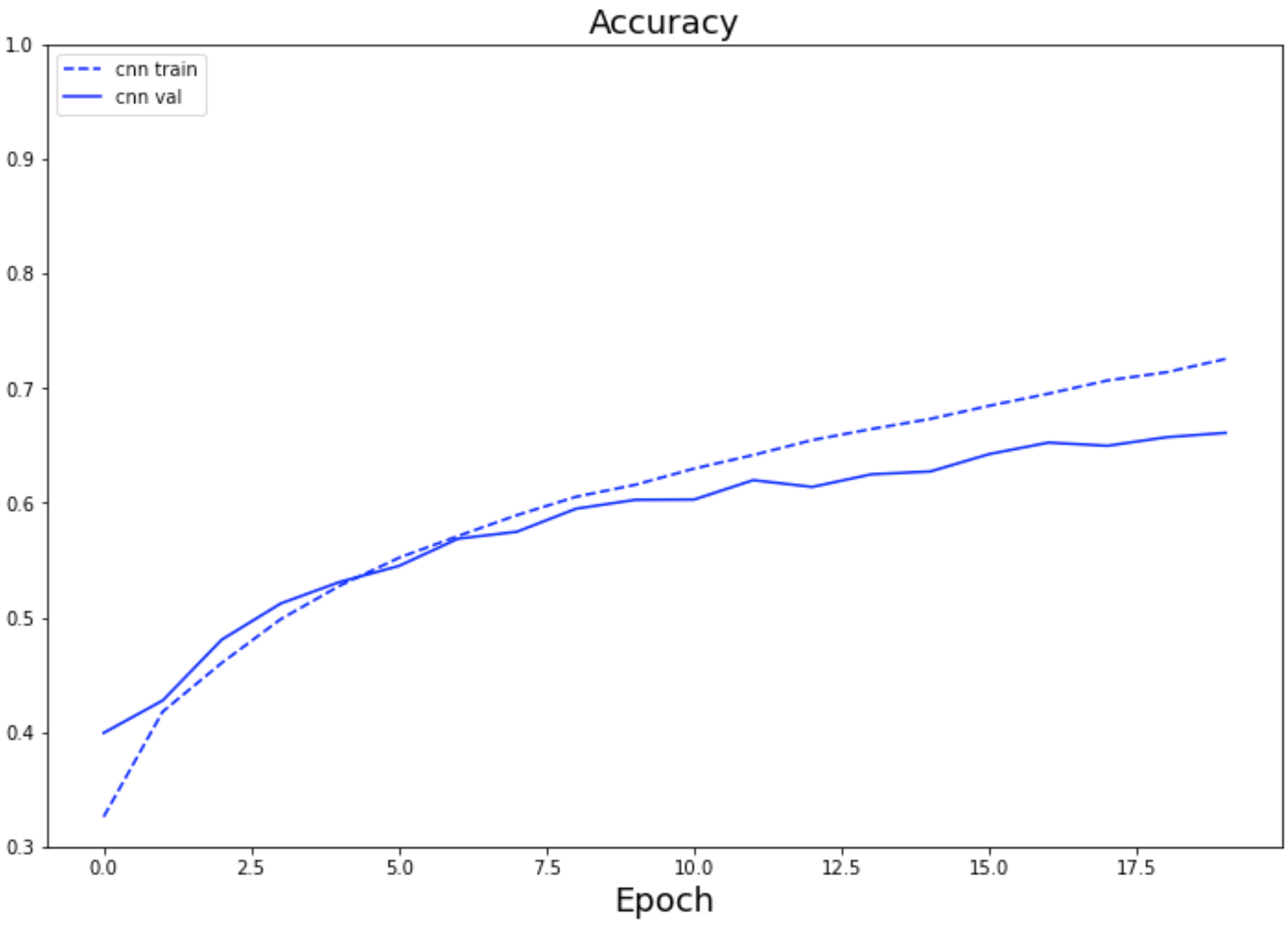
here, BN before applying the non-linearity

```python
for epoch in range(nepochs):
    # parameter update
    model.train()
    for batch, (X, Y) in enumerate(training_loader):
        pred = model(X)
        loss = cost_ce(pred, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# evaluation
model.eval()
...
```

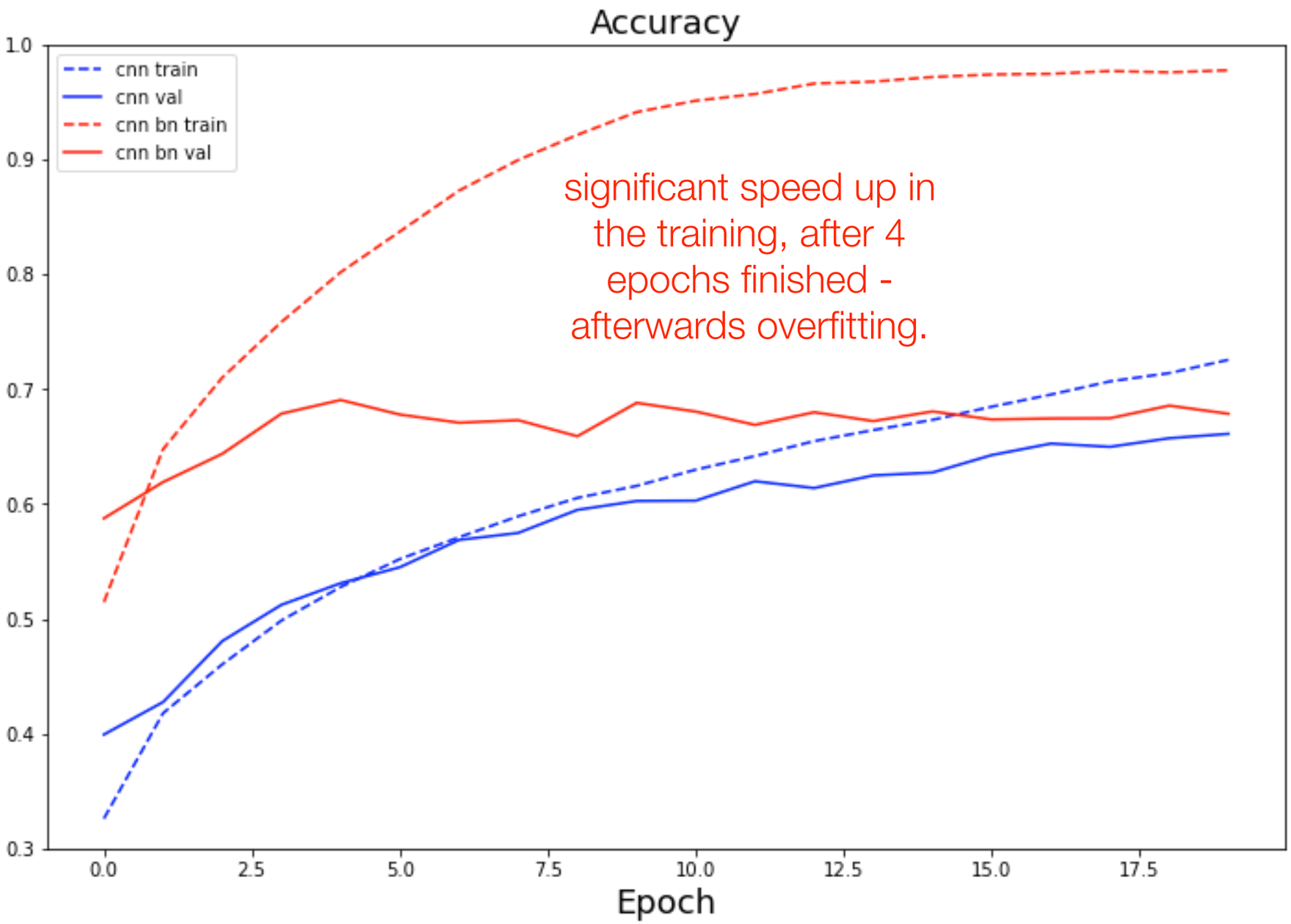
Different behavior of batch-norm at train and test/ validation time

Results



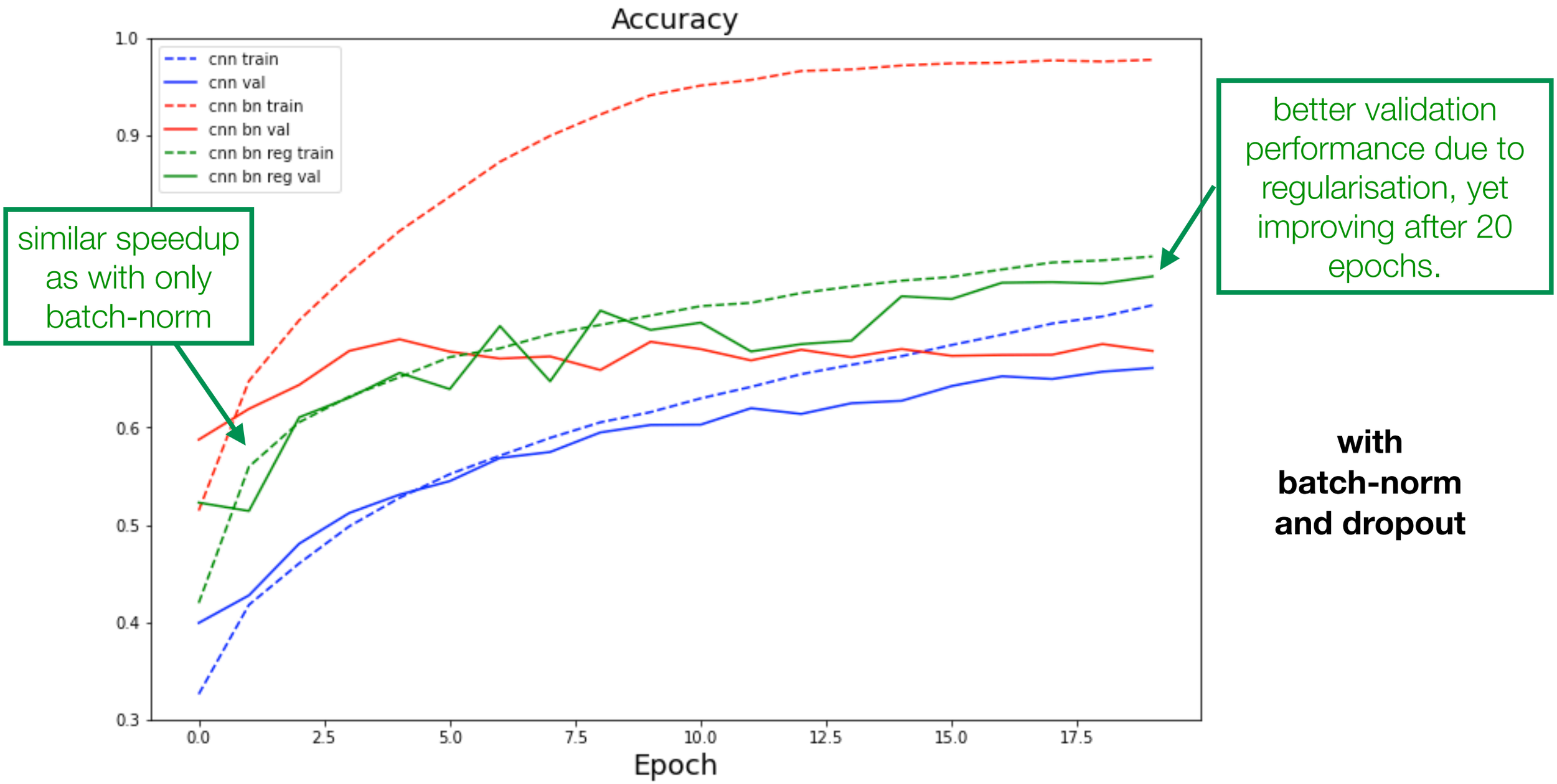
without batch-norm

Results



with batch-norm, after each CNN and Linear layer (except for the last)

Results



similar speedup as with only batch-norm

better validation performance due to regularisation, yet improving after 20 epochs.

with batch-norm and dropout

Batch Norm: Further Remarks

- Ongoing debate about where to apply batch norm - before/after activation fct?
  - Authors of original paper: before activation fct.
  - Often applied after activation fct.

- For implementation in Keras: Split up linear and non-linear layer operation.

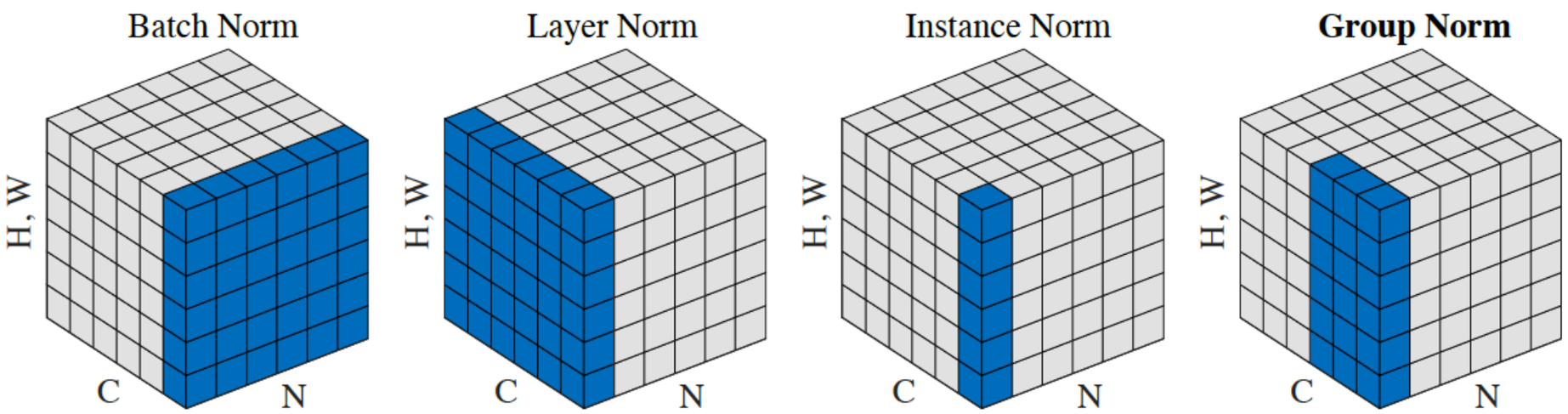
- For RNNs: Rather layer normalisation.

- Further reading:
  - Comprehensive blog post
  - First 20min of Goodfellow's lecture
  - Andrew Ilyas presentation

- Caution:
  - Batch size should not be too small for batch norm to be applicable
  - Different handling at training and test time may be delicate

Modifications, Generalisations

- Batch Norm: per batch - over channel, (H,W)
- Layer Norm: per layer and sample - over all activations.
  - Gives slightly worse improvements than Batch Norm, but can be performed per sample, i.e. similar behaviour in train and test.
- Instance Norm: per channel and sample - over (H,W)



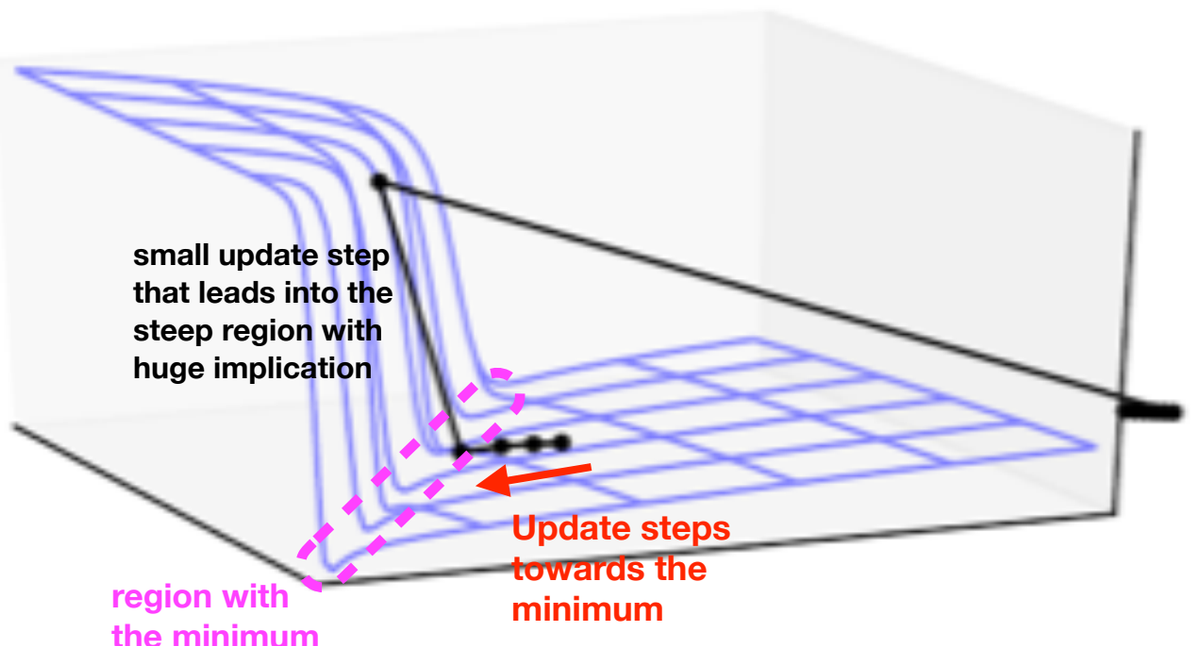
J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. CoRR, abs/1607.06450, 2016.
Y. Wu and K. He. Group normalization. CoRR, abs/1803.08494, 2018.

Exploding Gradients?

Gradient Clipping

- In case of *exploding gradients* : Clip gradients before update
  - clip length of gradient (**`clipnorm`**) or abs value of components (**`clipvalue`**).

- Intuition provided in the “Deep Learning” book (I. Goodfellow):
  Deep networks often have extremely steep regions resembling cliffs
  - as a result of the composite structure of the operations where weights get multiplied.
  At the cliffs, the gradient can be become very large.



often needed in RNNs
(see next week)

Gradient Clipping: Implementation

- Tensorflow/Keras : Parameters of the optimiser
  - **`clipnorm`**, **`clipvalue`**

- PyTorch:

```
optimizer.zero_grad()
loss, hidden = model(data, hidden, targets)
loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), maxgradnorm)
optimizer.step()
```

Wrap-up

Stabilise and Improve Training Performance

- Input Normalisation
- Non Saturating Activation Function: (leaky) ReLU
- Xavier / He - Initialisation
- Batch Normalisation

PW 08

Demonstrate the effect of

- proper parameter initialisation
- batchnorm

on the training performance (speed).