Deep Learning

Lecture 3: Some hyperparameters, regularization techniques and practical recommendations

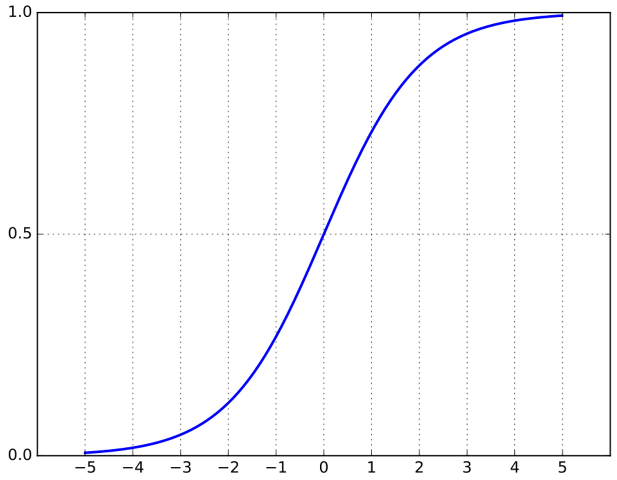
Prof. Stéphane Gaïffas
https://stephanegaiffas.github.io

Agenda for today

1. Activation functions
2. Output units and losses
3. Weight initialization
4. Regularization by penalization
5. Regularization by Dropout
6. Batch normalization, layer normalization
7. Early stopping
8. Final practical recommendations

Activation functions

Sigmoid



sigmoid activation

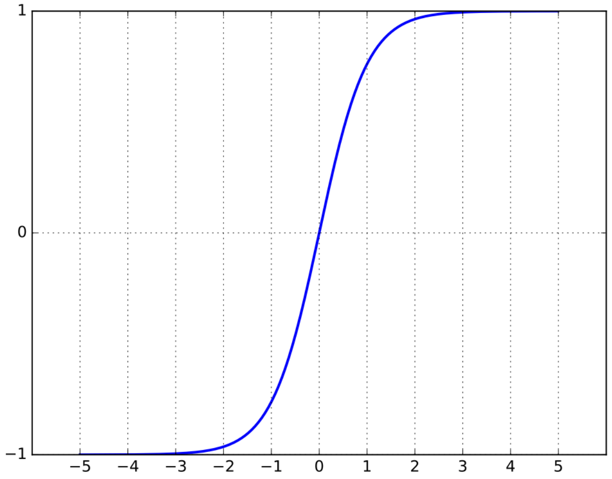
Sigmoid function

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

Comments

- Sigmoid is not centered at zero
- "Saturated" function: $\Rightarrow$ gradient killer
- Exp more computationally expensive (although negligible in general)

Tanh



$\tanh$ *activation*

Hyperbolic tangent function

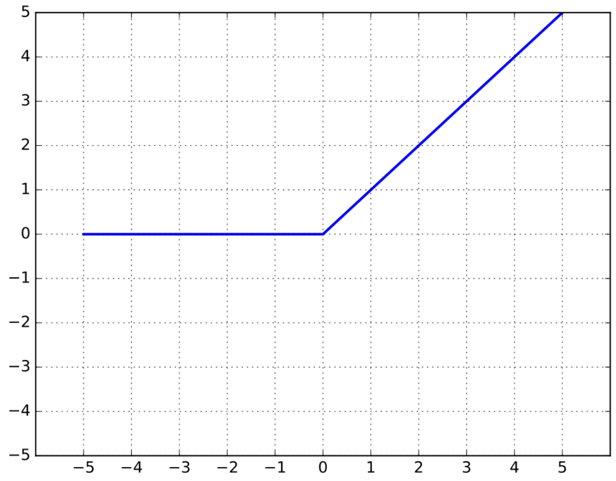
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

It's a rescaling in $[-1, 1]$ of the sigmoid

Comments

- Centered at zero
- "Saturated" function: $\Rightarrow$ gradient killer
- Exp more computationally expensive (although negligible in general)

Rectified Linear Unit (ReLU)



ReLU activation

ReLU (positive part)

$$\mathrm{ReLU}(x) = \max(x, 0)$$

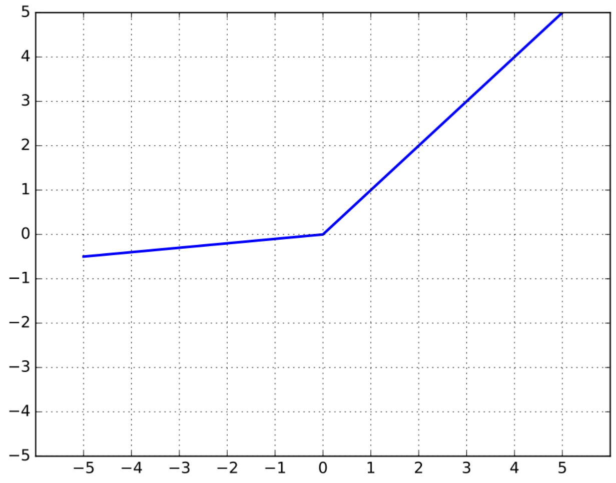
Comments

- Introduced in [1]
- Not a saturated function (piecewise linear) and sparse output
- Typically leads to architectures that can be trained faster than sigmoid or tanh (since it mitigates vanishing gradient)
- Related to biology [2], Biologically plausible

———

[1]: "Imagenet classification with deep convolutional neural networks", Krizhevsky et al. 2012
[2]: "Deep sparse rectifier neural networks", Glorot, Bordes, et al. 2011

Variations around ReLU



Leaky ReLU activation with $\alpha = 0.1$

Activations of the form

$$x \mapsto \max(\alpha x, x)$$

Leaky ReLU

- It's $\alpha = 0.1$
- Introduced in [1]

Absolute Value Rectification

- It's $\alpha = -1$
- Introduced in [2]

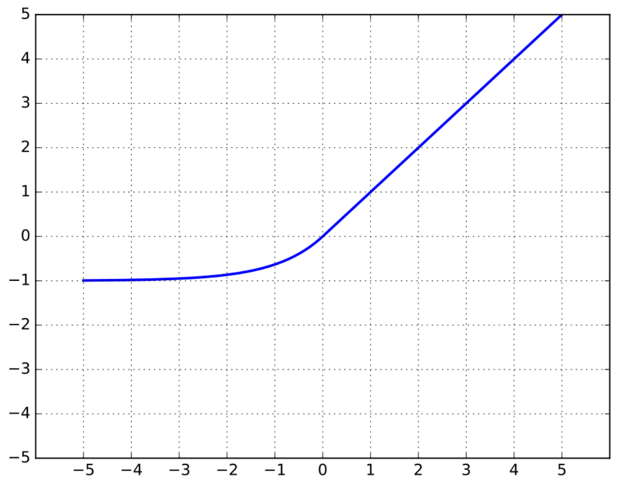
Parametric ReLU

- α is optimized (learned activation)
- Introduced in [3]

———

[1]: "Rectifier nonlinearities improve neural network acoustic models", Maas et al. 2013
[2]: "What is the best multi-stage architecture for object recognition?", Jarrett et al. 2009
[3]: "Empirical evaluation of rectified activations in convolutional network", Xu et al. 2015

ELU



ELU activation with $\alpha = 1$

Exponential Linear Unit

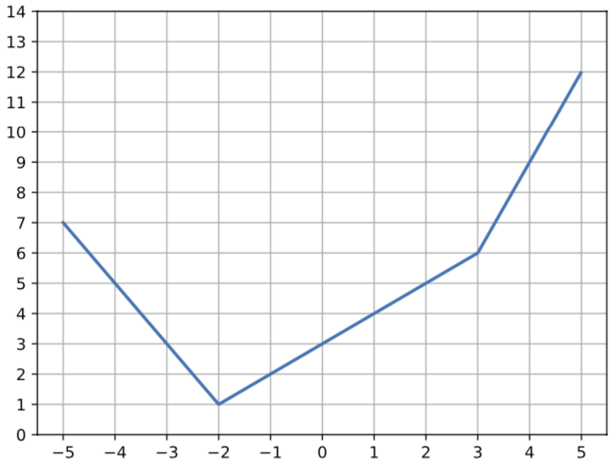
$$x \mapsto \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

- Negative saturation regime, closer to zero-mean output
- α typically set to 1.0
- Robustness to noise [1]

———

[1]: Fast and accurate deep network learning by exponential linear units (elus)", Clevert et al. 2015

Maxout



Maxout activation with $k = 3$

Maxout unit

Replaces a dense layer

$$h = \mathrm{ReLU}(\mathbf{W}x + b)$$

where $\mathbf{W} \in \mathbb{R}^{q \times d}$ by

$$h = \max(\mathbf{W}x + \mathbf{B})$$

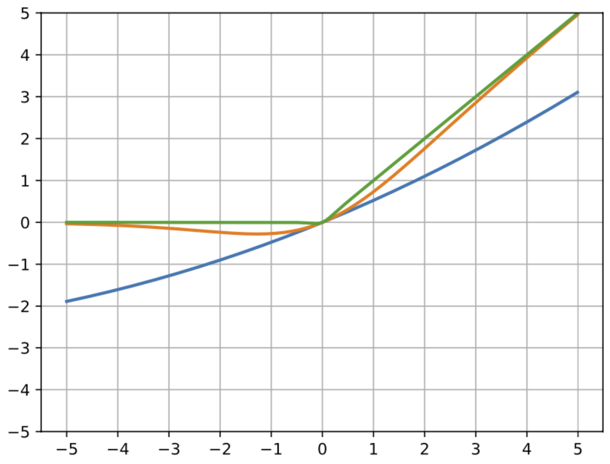
where $\mathbf{W} \in \mathbb{R}^{k \times q \times d}$ and $\mathbf{B} \in \mathbb{R}^{k \times q}$ and $\max$ if over the extra dimension (max of k coordinates)

- Number of parameters $\times k$
- Learns piecewise linear functions up to k pieces [1], [2]
- Generalizes ReLU and leaky ReLU with $k = 2$

---

[1] "Maxout networks", Goodfellow, Warde-Farley, et al. 2013
[2] "Deep maxout neural networks for speech recognition", Cai et al. 2013

Swish



Swish activation for $\alpha = 0.1, 1, 10$

Swish activation

$$x \mapsto \frac{x}{1 + e^{-\alpha x}}$$

- Introduced in [1]
- Interpolates between the ReLU and the identity
- Non-monotonic which seems to be an important feature

———

[1] "Searching for activation functions", Ramachandran et al. 2017

Conclusion on activation functions

- Use ReLU (or Swish ?)
- You can try Leaky ReLU, maxout, ELU
- You can try tanh, but do not expect too much
- Do not use sigmoid (unless you have reasons to)

Output units and losses

Output units and losses

Linear output unit

For multivariate regression with a label $y \in \mathbb{R}^K$

$$\widehat{y} = \mathbf{W}^\top h + b \quad \text{with loss} \quad \ell(y, \widehat{y}) = \|y - \widehat{y}\|_2^2$$

where $\mathbf{W} \in \mathbb{R}^{d \times K}$, $b \in \mathbb{R}^K$ and $h \in \mathbb{R}$.

It's a standard multivariate least-squares regression using h as input features

Sigmoid output unit

For binary classification with a label $y \in \{-1, 1\}$

$$\widehat{y} = \sigma(w^\top h + b) \quad \text{with loss} \quad \ell(y, \widehat{y}) = \log(1 + e^{-y\widehat{y}})$$

where $w \in \mathbb{R}^d$, $b \in \mathbb{R}$ and $h \in \mathbb{R}^d$.

It's a standard logistic regression using h as input features

Output units and losses

Softmax output unit

For K-class classification with a label $y \in \{1, \ldots, K\}$

$$\hat{y} = \text{softmax}(\mathbf{W}^\top h + b) \quad \text{where} \quad \ell(y, \hat{y}) = \text{crossentropy}(y, \hat{y})$$

where $\mathbf{W} \in \mathbb{R}^{d \times K}$ and $b \in \mathbb{R}^K$ and where

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{k=1}^K e^{z_k}} \cdots \frac{e^{z_K}}{\sum_{k=1}^K e^{z_k}} \right]$$

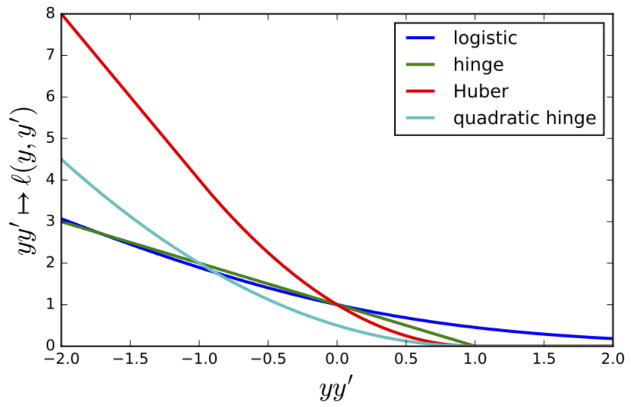
and

$$\text{crossentropy}(y, \hat{y}) = -\sum_{k=1}^{K} \mathbf{1}(k = y) \log(\hat{y}_k)$$

It's a standard softmax regression also known as multi-class logistic regression using h as input features

Output units and losses

- The cross-entropy is very popular
- Least-squares loss should not be used with softmax outputs [1]
- There's a bunch of other losses (mean absolute error, hinge loss, Huber loss, ad-hoc losses in computer vision, loss combinations, etc.)
- More complex models than softmax or least-square error: conditional Gaussian mixtures (multimodal y)



Some losses for binary classification

---

[1]: Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition", Bridle 1990

Weight initialization

Weight initialization

- In convex problems, provided a good learning rate, convergence is guaranteed regardless of the initial parameter values.
- In the non-convex regime, initialization is much more important!
- Little is known on the mathematics of initialization strategies of neural networks
- What is known: initialization should break symmetry and the scale of weights is important
- Have a look at: https://www.deeplearning.ai/ai-notes/initialization/

Bad ideas

- set all weights and biases to the same value (for instance 0): all neurons are going to be the same, in each iteration, too much symmetry
- small random numbers: might work for small networks but not for deeper networks, we end-up with no activation at all
- big random numbers: saturating phenomenon and overflow

Variances in the forward pass

- A first strategy is to initialize the network parameters such that activations preserve the same variance across layers
- Intuitively, this ensures that the information keeps flowing during the forward pass, without reducing or magnifying the magnitude of input signals exponentially

Let us assume that

- we are in a linear regime at initialization (e.g., the positive part of a ReLU or the middle of a sigmoid),
- weights w_{ij}^l are initialized i.i.d,
- biases b_l are initialized to be 0,
- input features are i.i.d, with a variance denoted as $\mathbb{V}[x]$.

Variances in the forward pass

Then, the variance of the activation h_i^l of unit i in layer l is

$$\mathbb{V}\left[h_i^l\right] = \mathbb{V}\left[\sum_{j=1}^{q_{l-1}} w_{ij}^l h_j^{l-1}\right] = \sum_{j=1}^{q_{l-1}} \mathbb{V}[w_{ij}^l]\mathbb{V}[h_j^{l-1}]$$

where q_l is the width of layer l and $h_j^0 = x_j$ for all $j = 1, \ldots, p$. Since the weights w_{ij}^l at layer l share the same variance $\mathbb{V}[w^l]$ and the variance of the activations in the previous layer are the same, we can drop the indices and write

$$\mathbb{V}[h^l] = q_{l-1}\mathbb{V}[w^l]\mathbb{V}[h^{l-1}].$$

Therefore, the variance of the activations is preserved across layers when

$$\mathbb{V}[w^l] = 1/q_{l-1} \quad \forall l.$$

This condition is enforced in LeCun's uniform initialization, which is defined as

$$w_{ij}^l \sim \text{Uniform}\left(\left[-\sqrt{3/q_{l-1}}, \sqrt{3/q_{l-1}}\right]\right).$$

Variances in the backward pass

A similar idea can be applied to ensure that the gradients flow in the backward pass (without vanishing nor exploding), by maintaining the variance of the gradient with respect to the activations fixed across layers.

Under the same assumptions as before,

$$\begin{aligned}\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h_i^l}\right] &= \mathbb{V}\left[\sum_{j=1}^{q_{l+1}} \frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}} \frac{\partial h_j^{l+1}}{\partial h_i^l}\right] \\ &= \mathbb{V}\left[\sum_{j=1}^{q_{l+1}} \frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}} w_{j,i}^{l+1}\right] \\ &= \sum_{j=1}^{q_{l+1}} \mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}}\right] \mathbb{V}\left[w_{ji}^{l+1}\right]\end{aligned}$$

Variances in the backward pass

If we further assume that

- the gradients of the activations at layer l share the same variance
- the weights at layer $l+1$ share the same variance $\mathbb{V}\left[w^{l+1}\right]$,

then we can drop the indices and write

$$\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h^l}\right] = q_{l+1}\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h^{l+1}}\right]\mathbb{V}\left[w^{l+1}\right].$$

Therefore, the variance of the gradients with respect to the activations is preserved across layers when

$$\mathbb{V}\left[w^l\right] = \frac{1}{q_l} \quad \forall l$$

Xavier initialization

We have derived two different conditions on the variance of w^l:

$$\mathbb{V}\left[w^l\right] = \frac{1}{q_{l-1}} \quad \text{and} \quad \mathbb{V}\left[w^l\right] = \frac{1}{q_l}$$

A compromise is the Xavier initialization, which initializes w^l randomly from a distribution with variance

$$\mathbb{V}\left[w^l\right] = \frac{1}{\frac{q_{l-1}+q_l}{2}} = \frac{2}{q_{l-1}+q_l}.$$

For example, normalized initialization is defined as

$$w_{ij}^l \sim \text{Uniform}\left(\left[-\sqrt{\frac{6}{q_{l-1}+q_l}}, \sqrt{\frac{6}{q_{l-1}+q_l}}\right]\right).$$

Examples of initializations

Xavier initialization [1]

$$w_{ij}^l \sim \mathrm{Uniform}\left(\left[-\sqrt{\frac{6}{q_{l-1}+q_l}}, \sqrt{\frac{6}{q_{l-1}+q_l}}\right]\right).$$

Gaussian initialization [2]

Initialize bias to zero and weights randomly using

$$w_{ij}^l \sim \mathrm{Normal}\left(\frac{\sqrt{2}}{q_{l-1}}\right)$$

LeCun's uniform initialization

$$w_{ij}^l \sim \mathrm{Uniform}\left(\left[-\sqrt{\frac{3}{q_{l-1}}}, \sqrt{\frac{3}{q_{l-1}}}\right]\right).$$

---

[1]: "Understanding the difficulty of training deep feedforward neural networks" Glorot and Bengio 2010
[2]: "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", He et al. 2015
[3]: "All you need is a good init", Mishkin and Matas 2015

Impact of a careful initialization

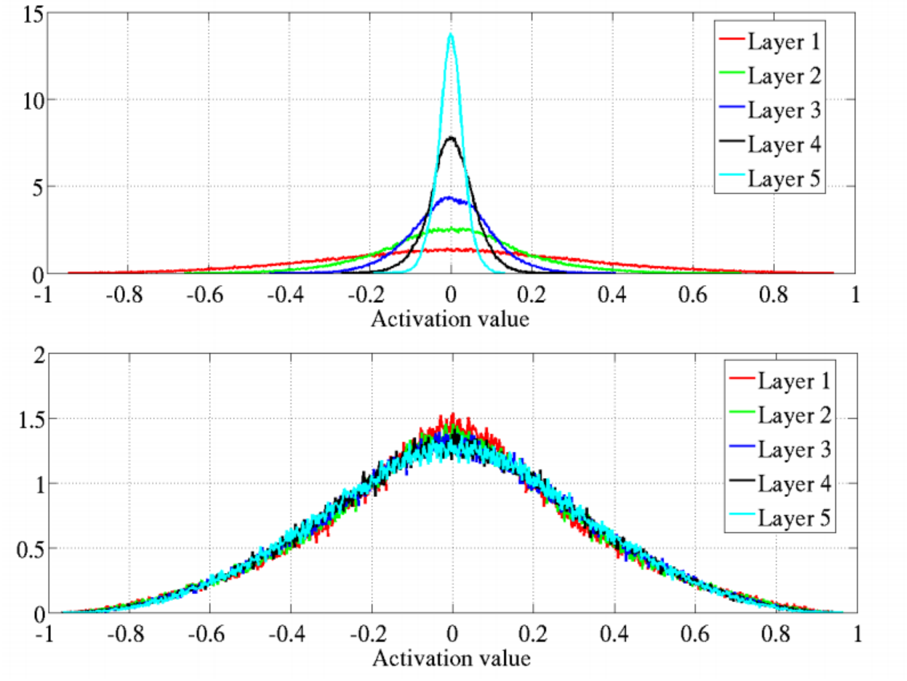


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

———

Impact of a careful initialization

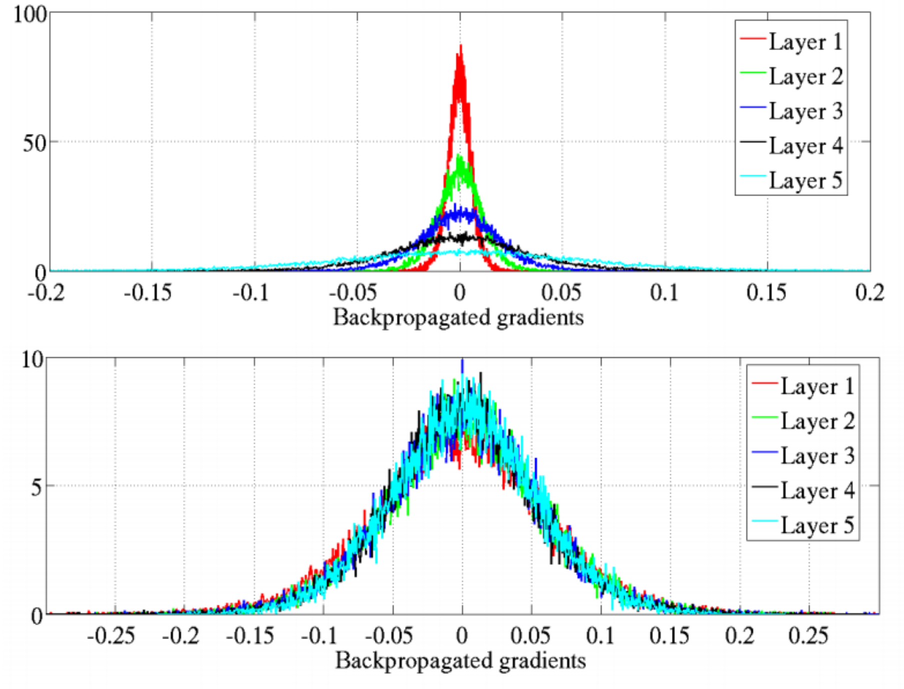
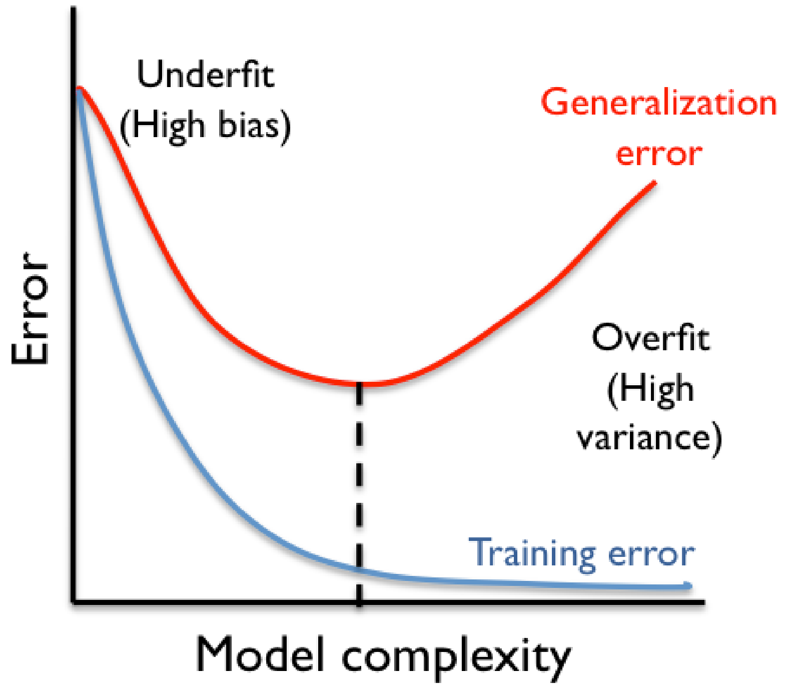


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Regularization by penalization

Regularization to avoid overfitting



- We usually need to impose some constraints over the parameters space to avoid overfitting

Avoid overfitting

Many different manners to avoid overfitting

- Penalization: Ridge, L1, etc. where we replace $F(\theta)$ by $F(\theta) + \mathrm{pen}(\theta)$ for some penalization function pen
- Weights sharing: such as with convolutional neural networks, where we reduce the parameters space by imposing to explicit constraints
- Dropout: kill at random some neurons during optimization, and predict with the full network
- Batch normalization: renormalize a layer inside a mini-batch, so that the network does not overfil on this particular batch
- Early stopping: stop the optimization procedure whenever the validation error does not decrease for a certain number of epochs

Penalization

A penalization can be applied on each layer individually or all the layers

Most standard one is Ridge, see [1] and [2] also called weight decay in deep learning literature where

$$\mathrm{pen}(\mathbf{W}) = \frac{\lambda}{2}\|\mathbf{W}\|_2^2$$

with a penalization hyper-parameter $\lambda > 0$. We can also use ℓ_1 penalization [3]

$$\mathrm{pen}(\mathbf{W}) = \lambda\|\mathbf{W}\|_1$$

or elastic-net [4]

$$\mathrm{pen}(\mathbf{W}) = \lambda\Big((1-\alpha)\frac{\lambda}{2}\|\mathbf{W}\|_2^2 + \alpha\|\mathbf{W}\|_1\Big)$$

with the extra hyper-parameter $\alpha \in [0, 1]$

---

[1]: "Ridge regression: Biased estimation for nonorthogonal problems", Hoerl and Kennard 1970
[2]: "Lecture notes on ridge regression", Wieringen 2015
[3]: "Regression shrinkage and selection via the lasso", Tibshirani 1996
[4]: "Regularization and variable selection via the elastic net", Zou and Hastie 2005

Regularization by Dropout

Dropout

Dropout [1] refers to dropping out units in a neural network: temporarily removing it from the network, along with its incoming and outgoing connections.

Typically, each unit is independently retained with a probability

- $p = 0.5$ for hidden units
- $p = 0.8$ for input units

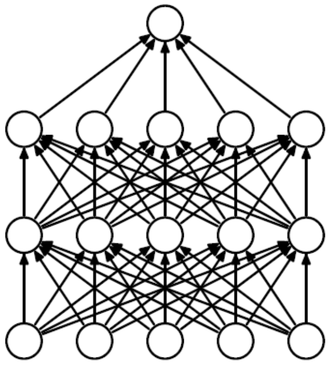
---

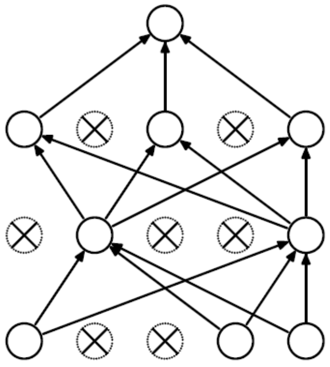
[1]: "Improving neural networks by preventing co-adaptation of feature detectors", Hinton et al. 2012

Dropout

During training: randomly remove units from the network. Update parameters as normal, leaving dropped-out units unchanged.

- For **each** sample in a mini-batch, we sample a thinned network by dropping out units
- Forward and back-propagation for that sample are done only on this thinned network. Any sample which does not use a parameter contributes a gradient of zero for that parameter

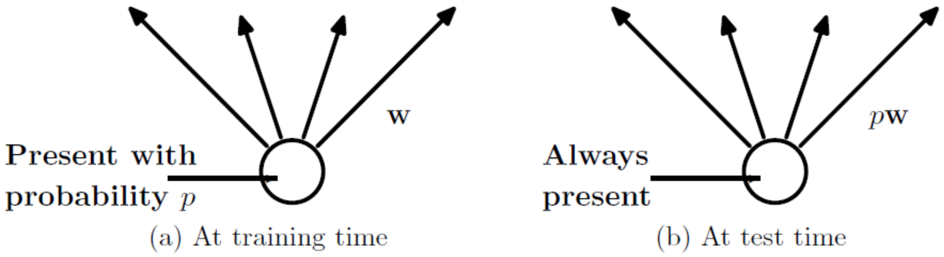

(a) Standard Neural Net


(b) After applying dropout.

Dropout

During testing (evaluation only): we need to account for this by rescaling.

- If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time
- This ensures that for any hidden unit the expected output (w.r.t the dropping distribution used at training time) is the same as the actual output at evaluation time



Dropout

Once again, with dropout we train as follows

Training: Inside one epoch, for a mini-batch of size m

- Sample m masks of iid Bernoulli random variables with probability p per node of the network (inner and input nodes but not output nodes).
- Usually $p = 0.5$ for hidden nodes and $p = 0.8$ for input nodes
- For each one of the m samples in the mini-batch:
    1. Do a forward pass on the masked network
    2. Compute back-propagation on the masked network
    3. Compute the mini-batch averaged gradient
- The optimizer updates the parameters as usual (nothing is changed here)

Prediction: Use all neurons in the network with the weights learned during training multiplied by the p used during training.

Dropout as an ensemble method

- Reminiscent of columns sub-sampling in random forests
- Roughly, Dropout averages different neural networks

Different can mean:

- Randomizing the training data (by dropping input units)
- Building and combining different network architectures

see [1], [2]

- Another strategy is to drop weights and not whole units [3]

Also, we can understand dropout as some form of penalization (next)

---

[1]: "Fast dropout training", Wang and Manning 2013
[2]: "Dropout: A simple way to prevent neural networks from overfitting", Srivastava et al. 2014"
[3]: "Regularization of neural networks using dropconnect", Wan et al. 2013

Dropout as penalization

- Consider linear regression with a matrix of features $\mathbf{X} \in \mathbb{R}^{n \times d}$ and labels $y \in \mathbb{R}^n$ and the least-squares objective $w \mapsto \|y - \mathbf{X}w\|_2^2$
- Here, Dropout drops inputs by replacing $\mathbf{X}$ by $\mathbf{B} \odot \mathbf{X}$ where $\mathbf{B}$ is a matrix containing iid $\mathrm{Bernoulli}(p)$ random variables.
- The objective would become

$$F(w) = \mathbb{E}_{\mathbf{B} \sim \mathrm{Bernoulli}(p)} \left[\|y - (\mathbf{B} \odot \mathbf{X})w\|_2^2 \right]$$

where $\odot$ is the Hadamard (entry-wise) product. An easy computation shows that this reduces to

$$F(w) = \|y - p\mathbf{X}w\|_2^2 + p(1-p)\|\Sigma w\|_2^2$$

where $\Sigma = \mathrm{diag}(\mathbf{X}^\top \mathbf{X})^{1/2}$ which rewrites by setting $w \leftarrow pw$

$$F(w) = \|y - \mathbf{X}w\|_2^2 + \frac{1-p}{p}\|\Sigma w\|_2^2$$

More details in "Dropout: A simple way to prevent neural networks from overfitting", Srivastava et al. 2014"

Batch normalization, layer normalization

Batch normalization

- The network converges faster if its input are scaled (mean, variance) and decorrelated [1]
- It's hard to decorrelate variables: requires to compute the whole covariance matrix [2]
- The previous weight initialization strategies rely on preserving the activation variance constant across layers, under the initial assumption that the input feature variances are the same, namely $\mathbb{V}[x_j] = \mathbb{V}[x]$ for any feature j

Aims and ideas

- Improving gradient flows
- Allowing higher learning rates
- Reducing strong dependence on initialization
- Related to regularization (maybe slightly reduces the need for Dropout)

———

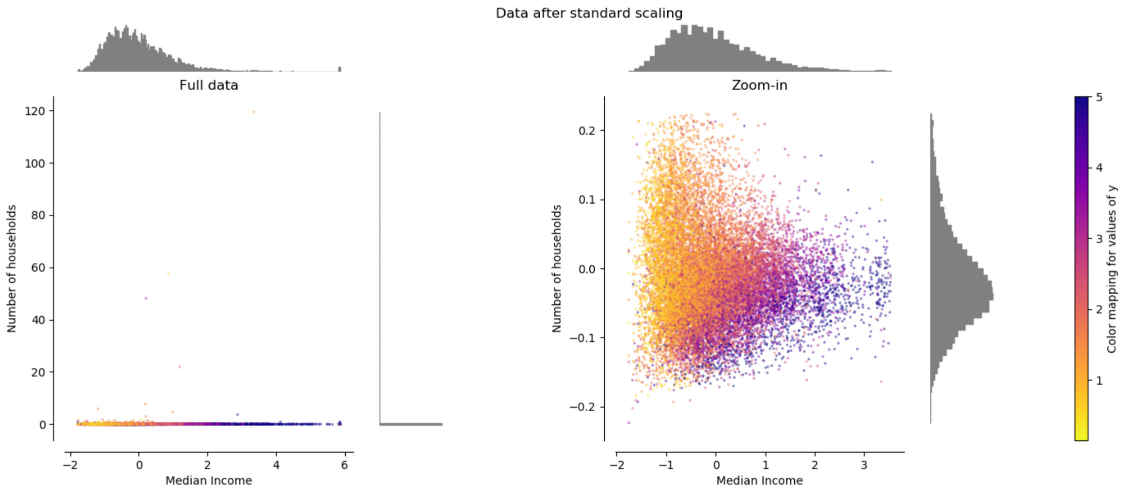
[1]: "Efficient backprop", LeCun et al. 1998
[2]: "Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015

The scaling of inputs is imposed by standardizing the input data feature-wise,

$$x' = (x - \widehat{\mu}) \odot \frac{1}{\widehat{\sigma}}$$

where

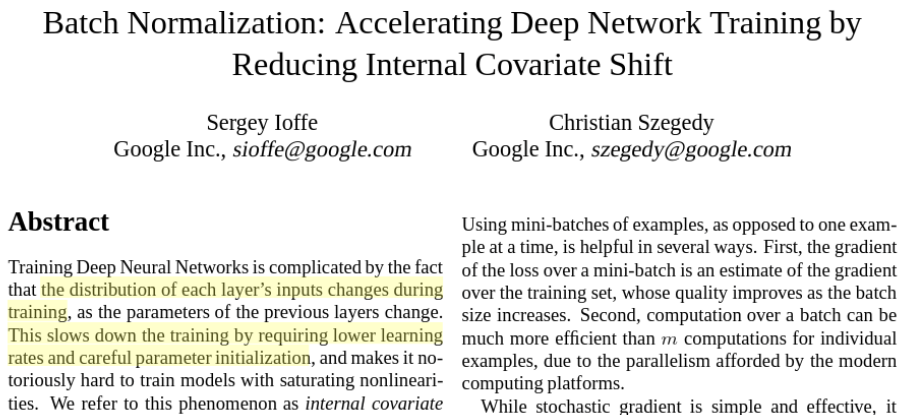
$$\widehat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \widehat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \widehat{\mu})^2}.$$



Credits: Scikit-Learn, Compare the effect of different scalers on data with outliers.

Batch normalization

- Maintaining proper statistics of the activations and derivatives is critical for training neural networks
- This constraint can be enforced explicitly during the forward pass by re-normalizing them
- Batch normalization was the first method introducing this idea



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate*

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

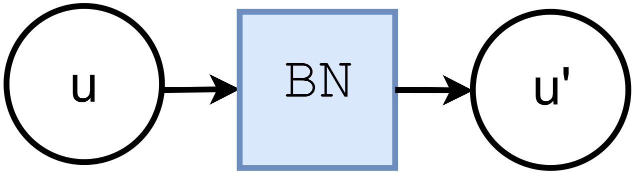
While stochastic gradient is simple and effective, it

———

"Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015

Batch normalization

- During training, batch normalization shifts and rescales according to the mean and variance estimated on the mini-batch
- During evaluation, it shifts and rescales according to the empirical moments estimated during training.
- Let $u_b \in \mathbb{R}^q$ be intermediate values computed at some location in the computational graph using a minibatch of training samples indexed by $b = 1, \ldots, B$
- Let us consider batch normalization applied following the node u



Batch normalization

During training, mean and variances are computed on each mini-batch B

$$\hat{\mu}_B = \frac{1}{B}\sum_{b=1}^{B} u_b \qquad \hat{\sigma}_B^2 = \frac{1}{B}\sum_{b=1}^{B}(u_b - \hat{\mu}_B)^2,$$

from which the standardized output $u'_b \in \mathbb{R}^q$ is computed as

$$u'_b = \gamma \odot (u_b - \hat{\mu}_B) \odot \frac{1}{\hat{\sigma}_B + \epsilon} + \beta$$

where $\gamma, \beta \in \mathbb{R}^q$ are trained parameters. Moreover, we maintain along the mini-batches global estimations $\hat{\mu}$ and $\hat{\sigma}^2$ of the mean and standard deviations

During inference, batch normalization shifts and rescales each component according to the global estimations computed during training

$$u' = \gamma \odot (u - \hat{\mu}) \odot \frac{1}{\hat{\sigma}} + \beta.$$

Batch normalization

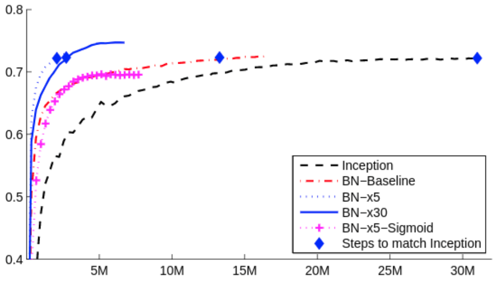
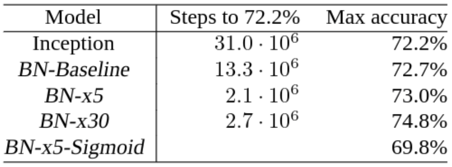


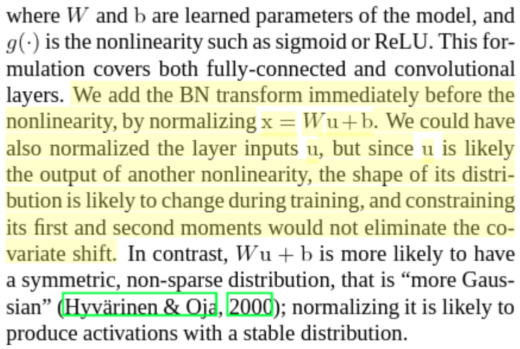
Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*



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.*

The position of batch normalization relative to the non-linearity is not clear



where W and b are learned parameters of the model, and $g(\cdot)$ is the nonlinearity such as sigmoid or ReLU. This formulation covers both fully-connected and convolutional layers. We add the BN transform immediately before the nonlinearity, by normalizing $\mathrm{x} = W\mathrm{u} + \mathrm{b}$. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $W\mathrm{u} + \mathrm{b}$ is more likely to have a symmetric, non-sparse distribution, that is "more Gaussian" (Hyvärinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution.

———
Credits: Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

Layer normalization

- Given a single input sample x, a similar approach can be applied to standardize the activations u across a layer instead of doing it over the samples of the mini-batch [1]
- Seems to be an important ingredient for transformer architectures [2]

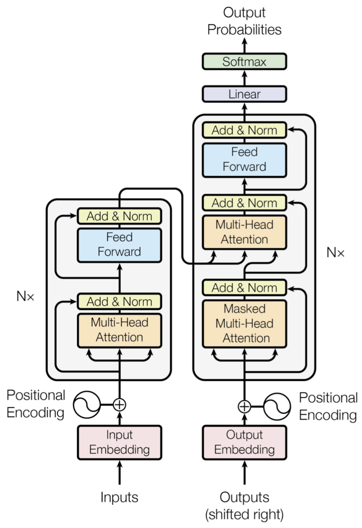
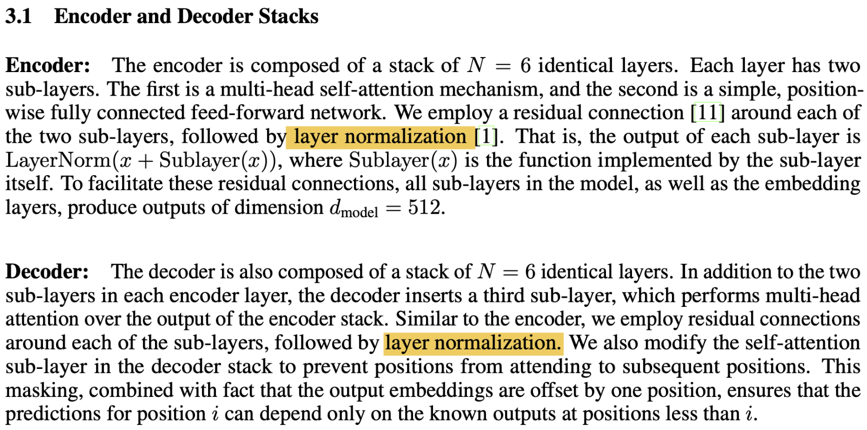


Figure 1: The Transformer - model architecture.



3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

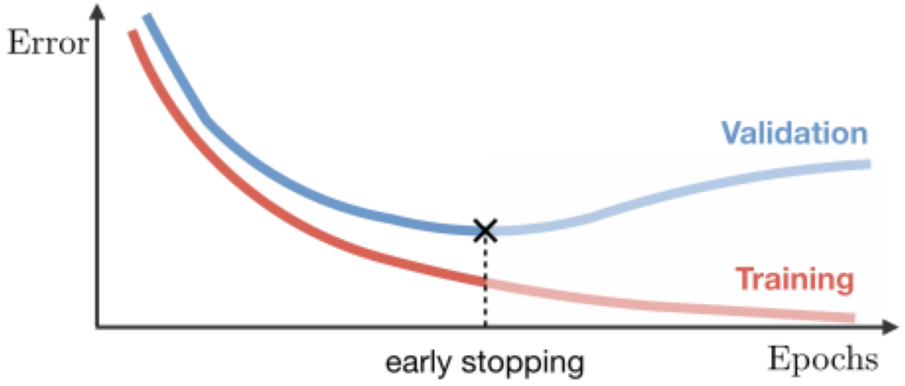
———

[1] "Layer Normalization", Ba, Kiros and Hinton 2016
[2] "Attention Is All You Need", Vaswani et al. 2017

Early stopping

Early stopping

- Checkpoint the model weights regularly (save everything on disk, disk storage is cheap...)
- Stop training when validation error goes up for a certain amount of epochs (patience parameter)
- Return the best checkpointed weights seen before, according to validation error and consider the corresponding number E of epochs as a hyperparameter
- Then, retrain on the full training data ($\text{training} \cup \text{validation}$) for E epochs, or wait until error matches the one observed at early stopping



Early stopping

Early stopping is an old idea

- "Three topics in ill-posed problems", Wahba 1987
- "A formal comparison of methods proposed for the numerical solution of first kind integral equations", Anderssen and Prenter 1981
- "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping", Caruana et al. 2001

But also an active area of research

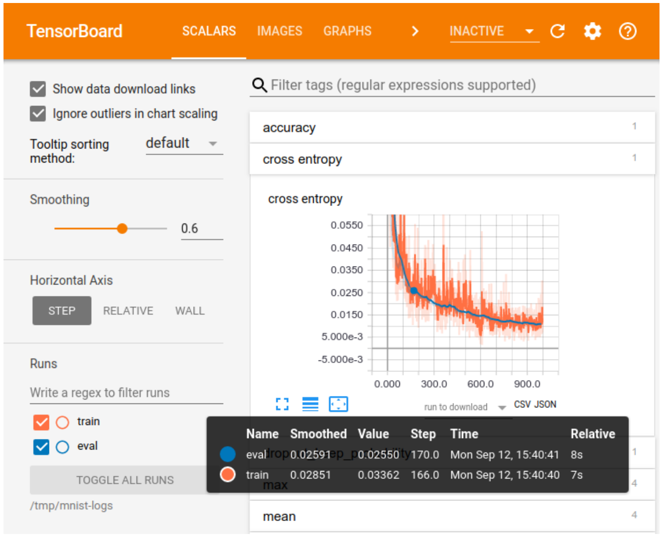
- "Adaboost is consistent", Bartlett and Traskin 2007
- "Boosting algorithms as gradient descent", Mason et al. 2000
- "On early stopping in gradient descent learning", Yao et al. 2007
- "Boosting with early stopping: Convergence and consistency", Zhang, Yu, et al. 2005
- "Early stopping for kernel boosting algorithms: A general analysis with localized complexities", Wei et al. 2017

Final practical recommendations

Training a large deep neural network is long, complex and sometimes confusing or counter-intuitive. This can take days, weeks or months !

A first step towards understanding, debugging, optimizing neural networks is to use visualization tools such as `TensorBoard` in order to:

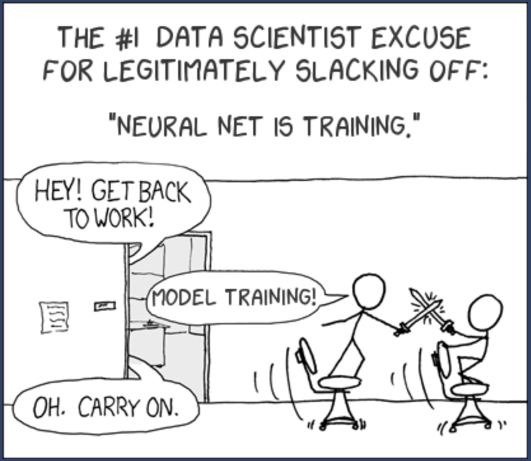
- plot losses and metrics
- visualizing computational graphs
- show additional data as the network is being trained (activation norms, etc.)



Name	Smoothed	Value	Step	Time	Relative
eval	0.02591	0.02550	170.0	Mon Sep 12, 15:40:41	8s
train	0.02851	0.03362	166.0	Mon Sep 12, 15:40:40	7s

The main steps usually look like this:

1. Preprocess the data and standardize it (if not, you need good reasons)
2. Build the architecture: input, outputs, layers, identify hyperparameters (ideally, they are in a configuration file, `.yaml` or other)
3. Consider dummy or simple baselines and train them to have baseline metrics
4. Starting training for few epochs your network and see if the loss is reasonable and compare it to baselines
5. Try to overfit a subset (or the whole data if time permits). You should be able to overfit a small part of the data
6. Add regularization and check if the error on the training set increases (even a little bit)
7. Hyper-optimize to find the best learning rate, mini-batch size and other hyper-parameters and... wait !
8. Retrain using the whole data
9. Check test error


THE #1 DATA SCIENTIST EXCUSE FOR LEGITIMATELY SLACKING OFF:
"NEURAL NET IS TRAINING."
HEY! GET BACK TO WORK!
MODEL TRAINING!
OH. CARRY ON.

Thank you !