Computer vision and deep learning

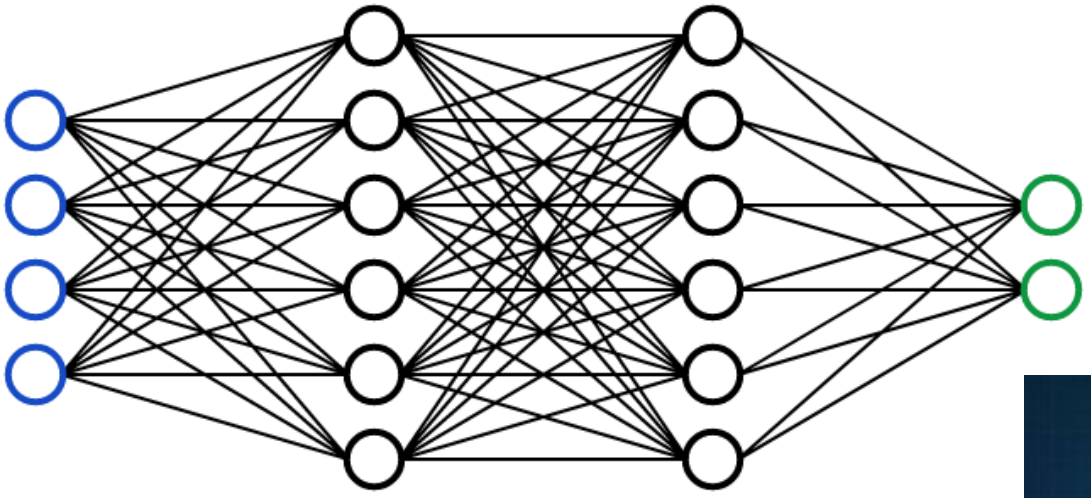
Lecture 5

Training a neural network

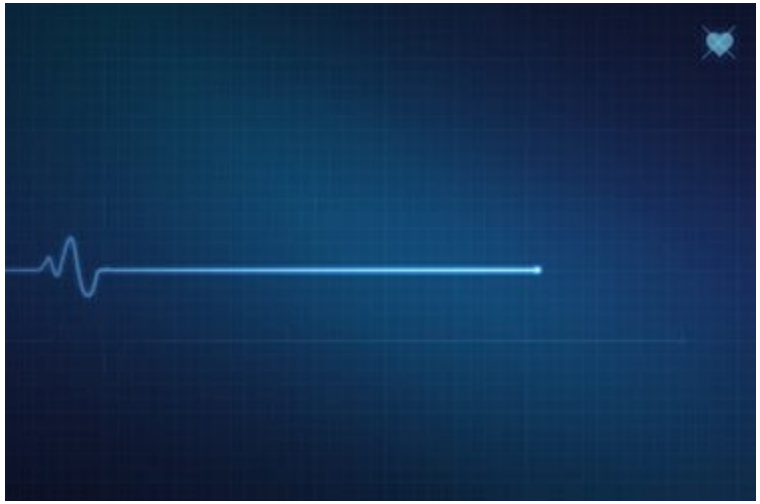
Training a neural network

Weights initialization

How **not** to initialize: zero weights



No symmetry breaking



Vanishing and exploding gradients

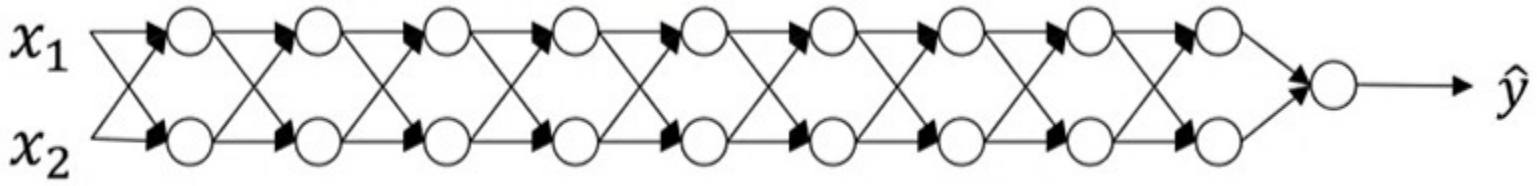


Image source: https://www.coursera.org/learn/deep-neural-network/lecture/C9iQO/vanishing-exploding-gradients

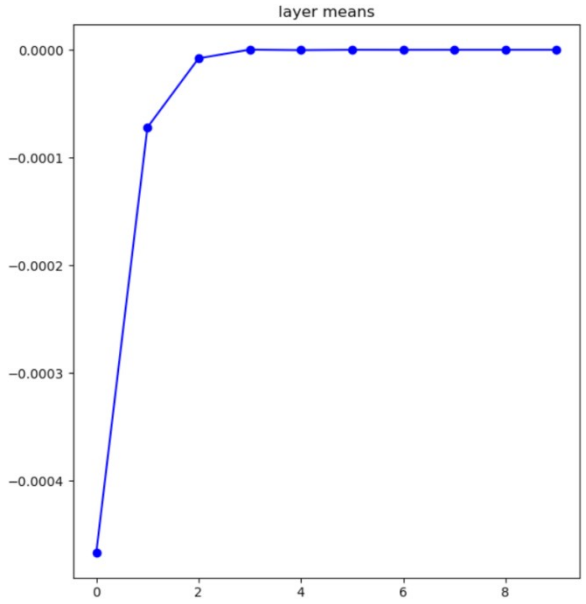
Weight initialization

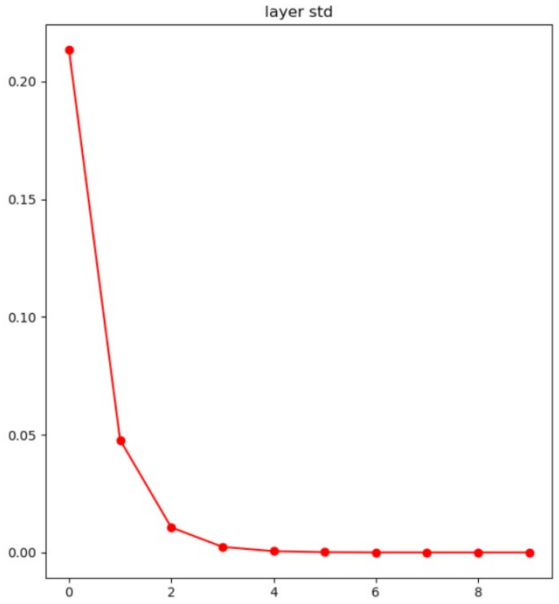
Example credit: Andrew Karpathy

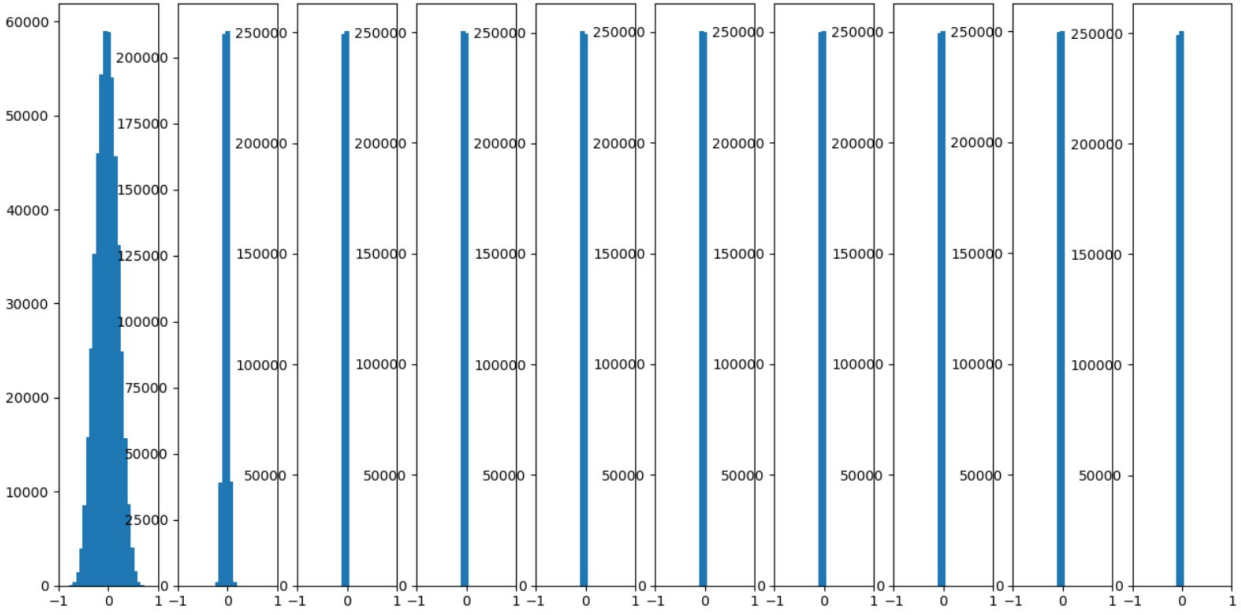
https://colab.research.google.com/drive/1CvCXZIevs6MvQnldyG2UHZevG0bKaeiu?usp=sharing

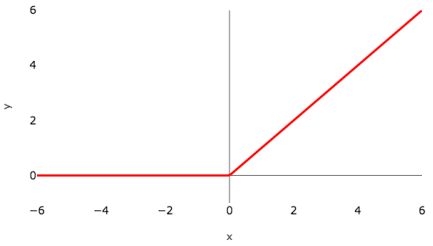
W = np.random.randn(in, out)*0.01

input layer had mean -0.001449 and stddev 0.999223
hidden layer 0 had mean 0.000289 and stddev 0.213667
hidden layer 1 had mean -0.000064 and stddev 0.047754
hidden layer 2 had mean -0.000001 and stddev 0.010667
hidden layer 3 had mean -0.000003 and stddev 0.002391
hidden layer 4 had mean 0.000000 and stddev 0.000535
hidden layer 5 had mean -0.000000 and stddev 0.000119
hidden layer 6 had mean 0.000000 and stddev 0.000027
hidden layer 7 had mean -0.000000 and stddev 0.000006
hidden layer 8 had mean 0.000000 and stddev 0.000001
hidden layer 9 had mean 0.000000 and stddev 0.000000









Xavier initialization

- *Understanding the difficulty of training deep feedforward neural networks* , Xavier Glorot Yoshua Bengio https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
- Random initialization from a distribution with a variance of:

- If inputs are roughly mean 0 and std 1, this initialization will also cause the outputs to have mean 0 and stddev 1

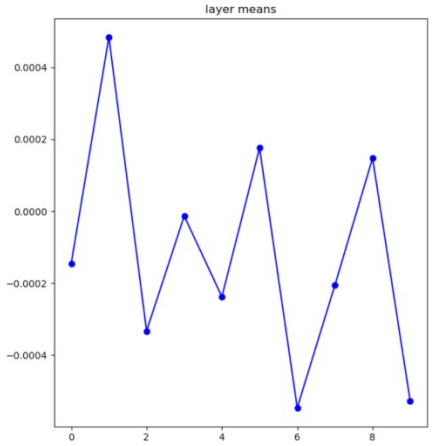
Additional reading:

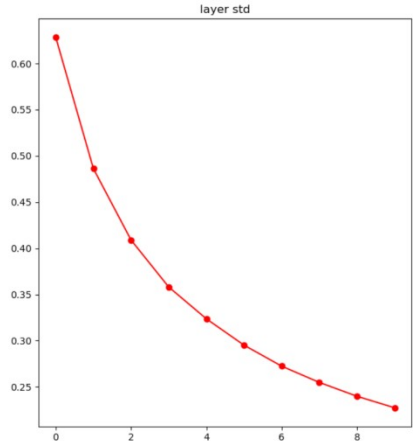
https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-functions-choose-wisely/

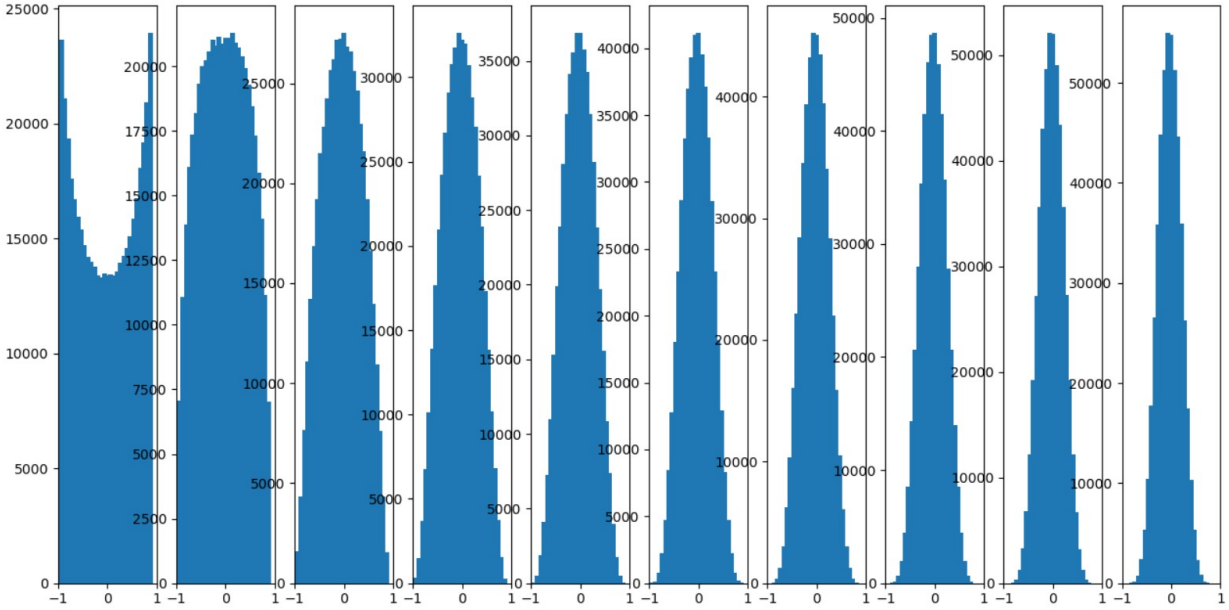
Xavier initialization

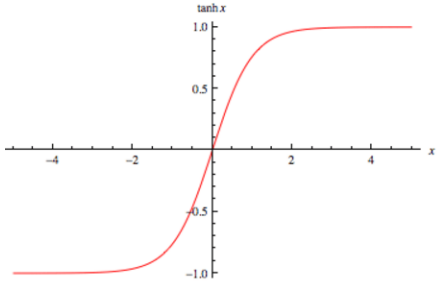
W = np.random.randn(in, out)/**np.sqrt(in)**

hidden layer 0 had mean -0.000145 and stddev 0.628485
hidden layer 1 had mean 0.000485 and stddev 0.486178
hidden layer 2 had mean -0.000333 and stddev 0.408851
hidden layer 3 had mean -0.000013 and stddev 0.358011
hidden layer 4 had mean -0.000238 and stddev 0.323565
hidden layer 5 had mean 0.000176 and stddev 0.295040
hidden layer 6 had mean -0.000547 and stddev 0.272295
hidden layer 7 had mean -0.000206 and stddev 0.254713
hidden layer 8 had mean 0.000148 and stddev 0.239710
hidden layer 9 had mean -0.000529 and stddev 0.227241





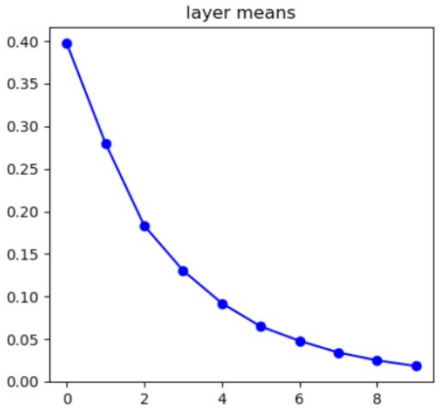


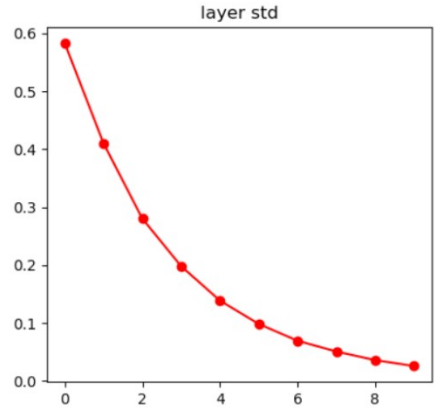


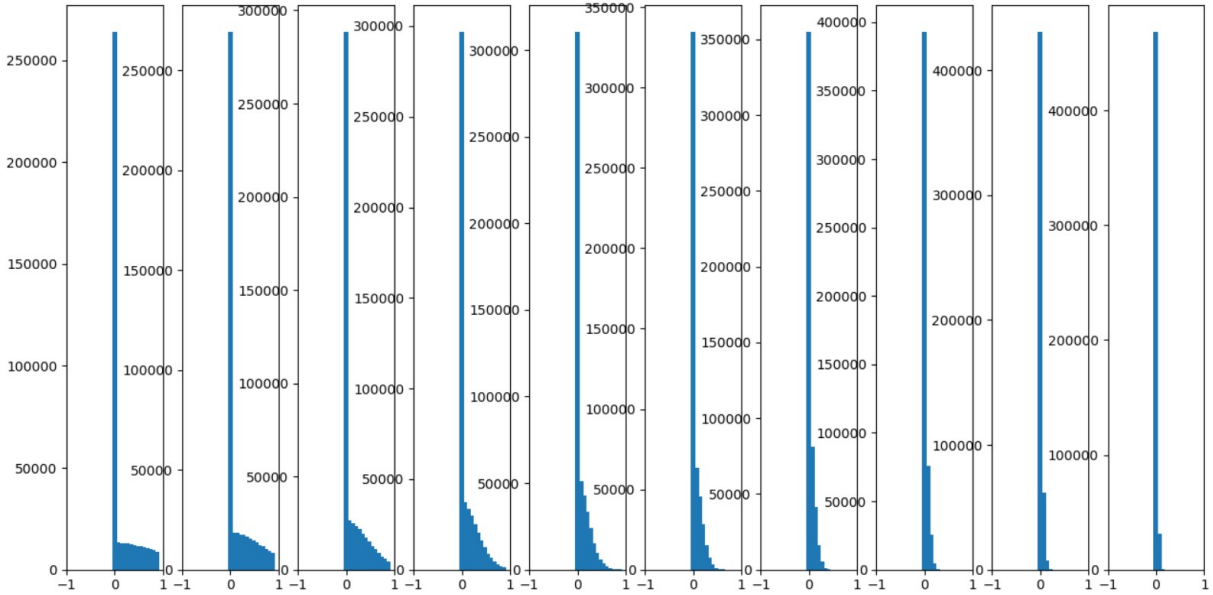
Xavier initialization

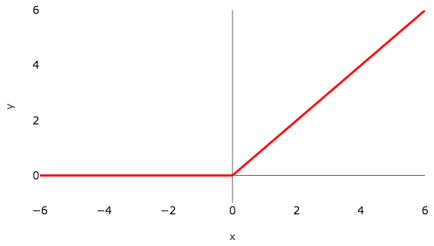
W = np.random.randn(in, out)/**np.sqrt(in)**

input layer had mean 0.000422 and stddev 1.000922
hidden layer 0 had mean 0.397685 and stddev 0.583384
hidden layer 1 had mean 0.280004 and stddev 0.410386
hidden layer 2 had mean 0.183151 and stddev 0.280291
hidden layer 3 had mean 0.130450 and stddev 0.198090
hidden layer 4 had mean 0.092056 and stddev 0.138666
hidden layer 5 had mean 0.064787 and stddev 0.098349
hidden layer 6 had mean 0.047867 and stddev 0.069325
hidden layer 7 had mean 0.034165 and stddev 0.050724
hidden layer 8 had mean 0.024928 and stddev 0.035902
hidden layer 9 had mean 0.018272 and stddev 0.025658





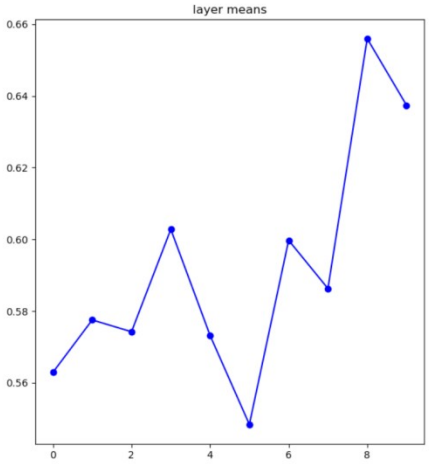


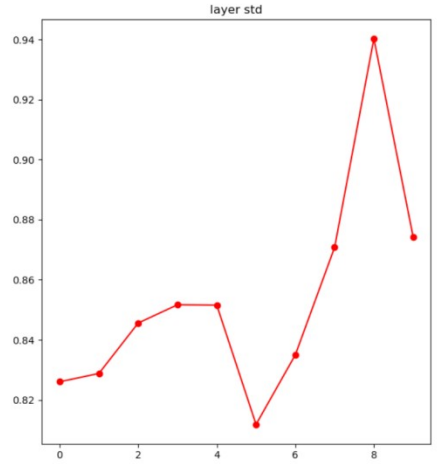


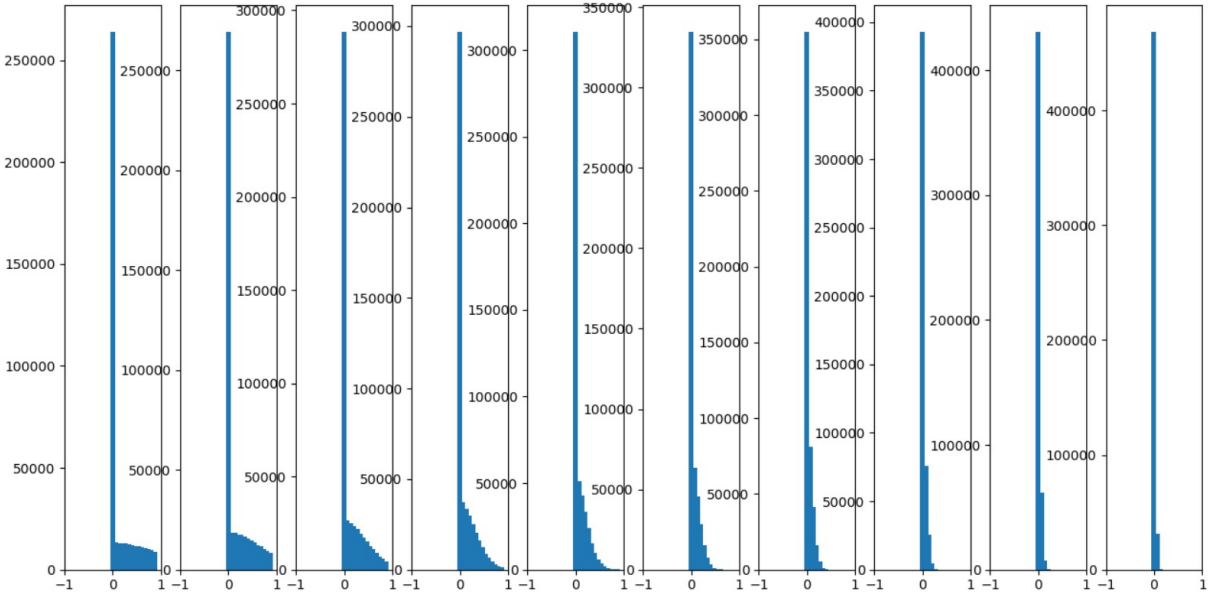
He initialization

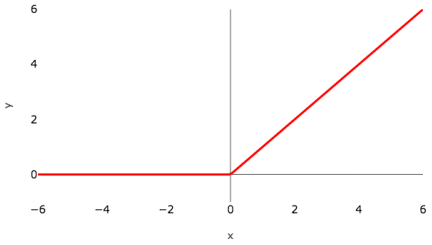
W = np.random.randn(in, out)/**np.sqrt(in/2)**

input layer had mean -0.001160 and stddev 1.000333
hidden layer 0 had mean 0.562934 and stddev 0.826029
hidden layer 1 had mean 0.577537 and stddev 0.828810
hidden layer 2 had mean 0.574250 and stddev 0.845599
hidden layer 3 had mean 0.602799 and stddev 0.851682
hidden layer 4 had mean 0.573220 and stddev 0.851573
hidden layer 5 had mean 0.548410 and stddev 0.811833
hidden layer 6 had mean 0.599740 and stddev 0.834997
hidden layer 7 had mean 0.586302 and stddev 0.870794
hidden layer 8 had mean 0.655965 and stddev 0.940311
hidden layer 9 had mean 0.637412 and stddev 0.874220









“Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”

https://arxiv.org/pdf/1502.01852.pdf

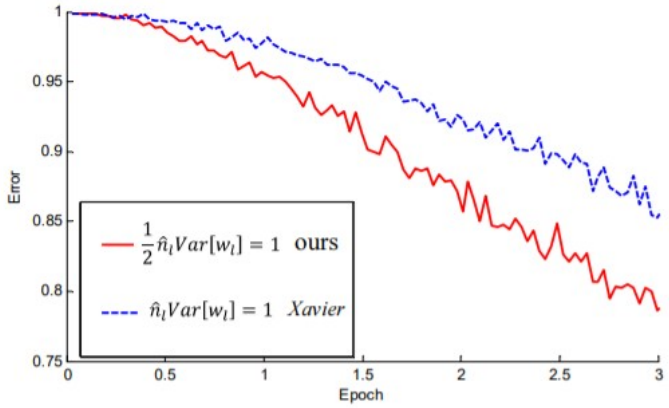


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and “*Xavier*” (blue) [7] lead to convergence, but ours starts reducing error earlier.

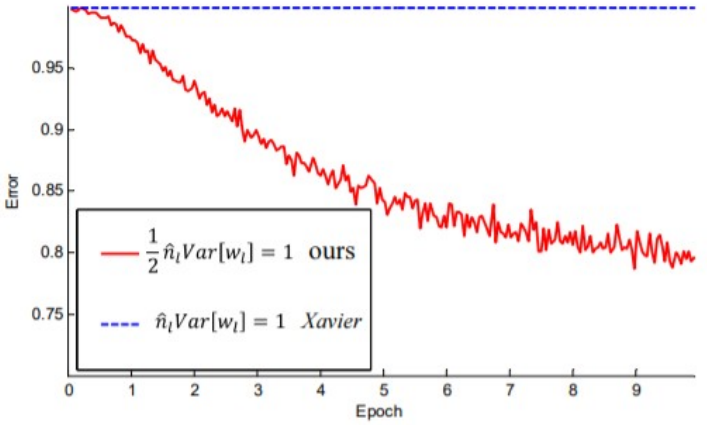


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But “*Xavier*” (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

Data driven initialization

- "All you need is a good init": https://arxiv.org/pdf/1511.06422.pdf

Algorithm 1 Layer-sequential unit-variance orthogonal initialization. L – convolution or full-connected layer, W_L - its weights, B_L - its output blob., Tol_{var} - variance tolerance, T_i – current trial, T_{max} – max number of trials.

Pre-initialize network with orthonormal matrices as in Saxe et al. (2014)
for each layer L **do**
  while $|Var(B_L) - 1.0| \geq Tol_{var}$ and $(T_i < T_{max})$ **do**
    do Forward pass with a mini-batch
    calculate $Var(B_L)$
    $W_L = W_L \,/\, \sqrt{Var(B_L)}$
  end while
end for

keras initialization

Conv2D class

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Dense class

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

- initializers
  - Overview
  - Constant
  - GlorotNormal
  - GlorotUniform
  - HeNormal
  - HeUniform
  - Identity
  - Initializer
  - LecunNormal
  - LecunUniform
  - Ones
  - Orthogonal
  - RandomNormal
  - RandomUniform
  - TruncatedNormal
  - VarianceScaling
  - Zeros

Training a neural network

Regularization

Regularization

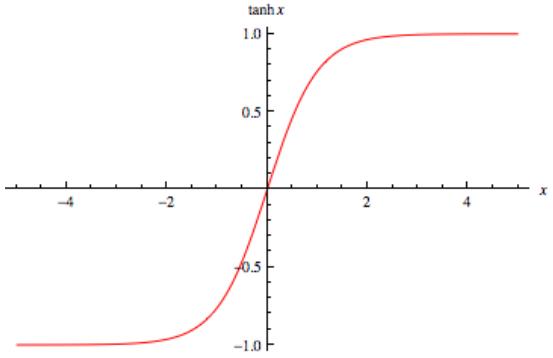
- Remember regularization techniques
  - L1 regularization
  - L2 regularization

Regularization

- Weight decay

Regularization

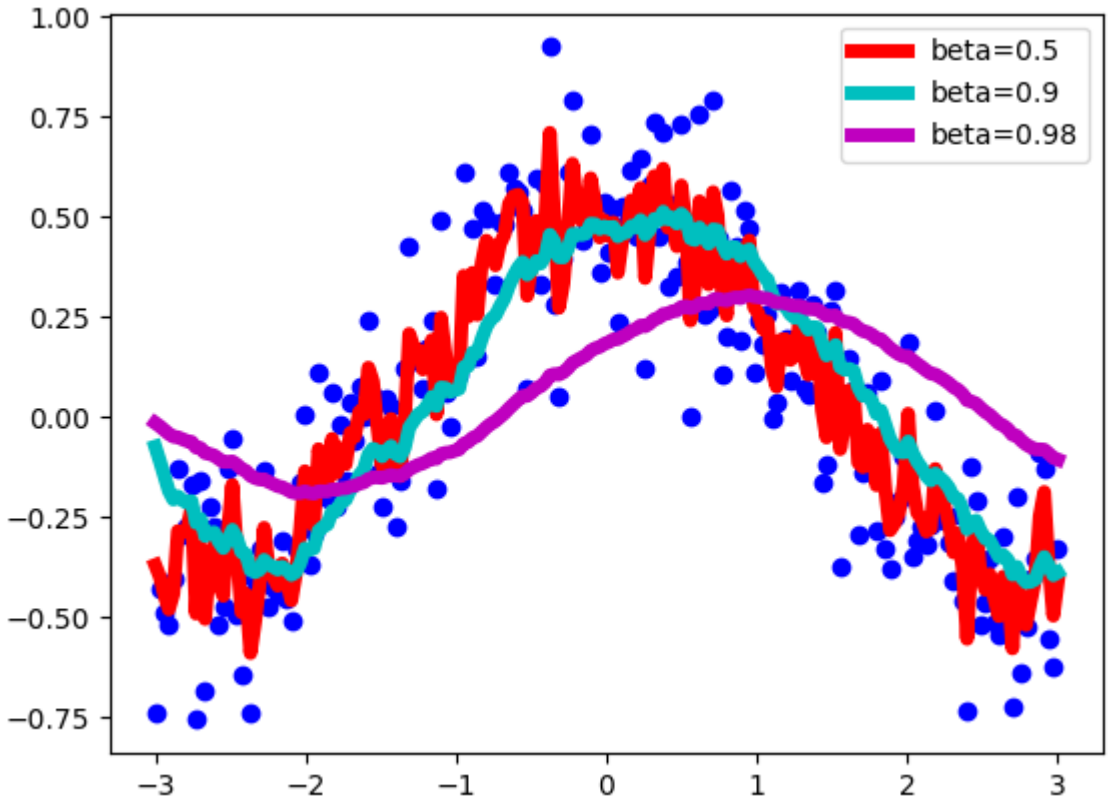
- Why does regularization reduce overfitting?



Exponential weighted averages

https://
colab.research.google.com/drive/1CTSctxtN1JDvGZOswpLlG9ZkC4-s0ym2?usp=sharing

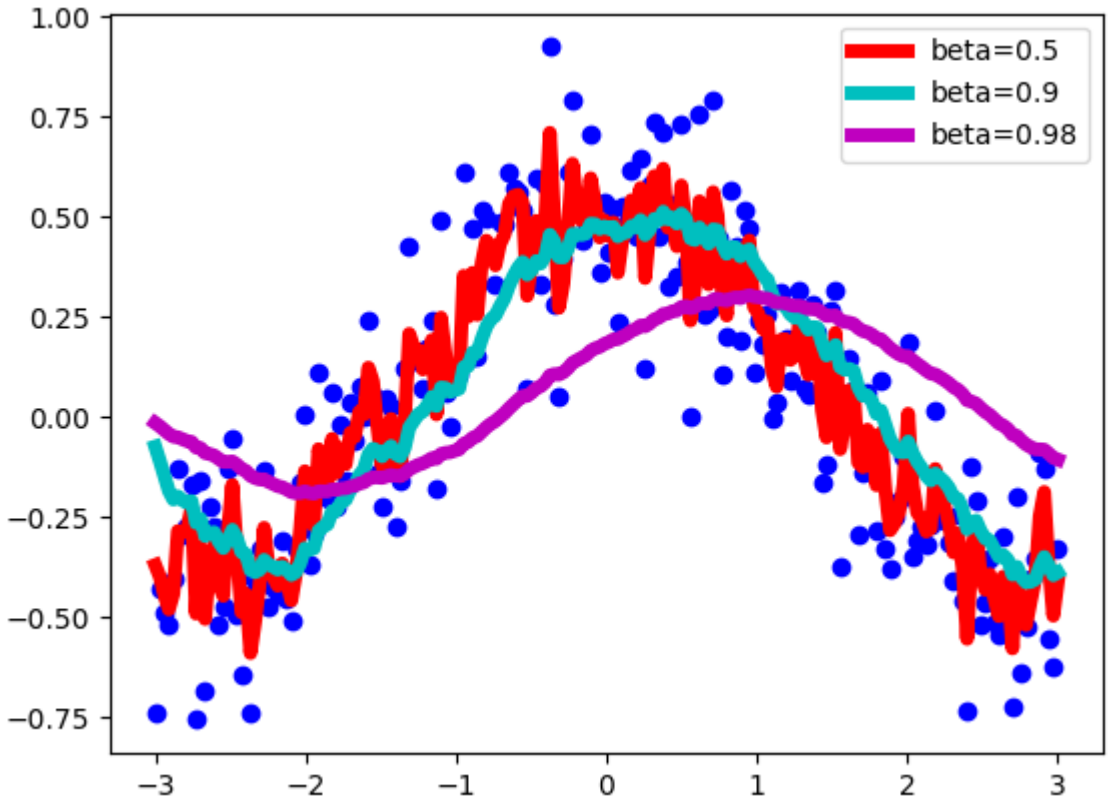
Exponential weighted averages



$$V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot x$$

https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

Exponential weighted averages



V_t approximates over: $\frac{1}{1-\beta}$ samples

https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

Exponential weighted average

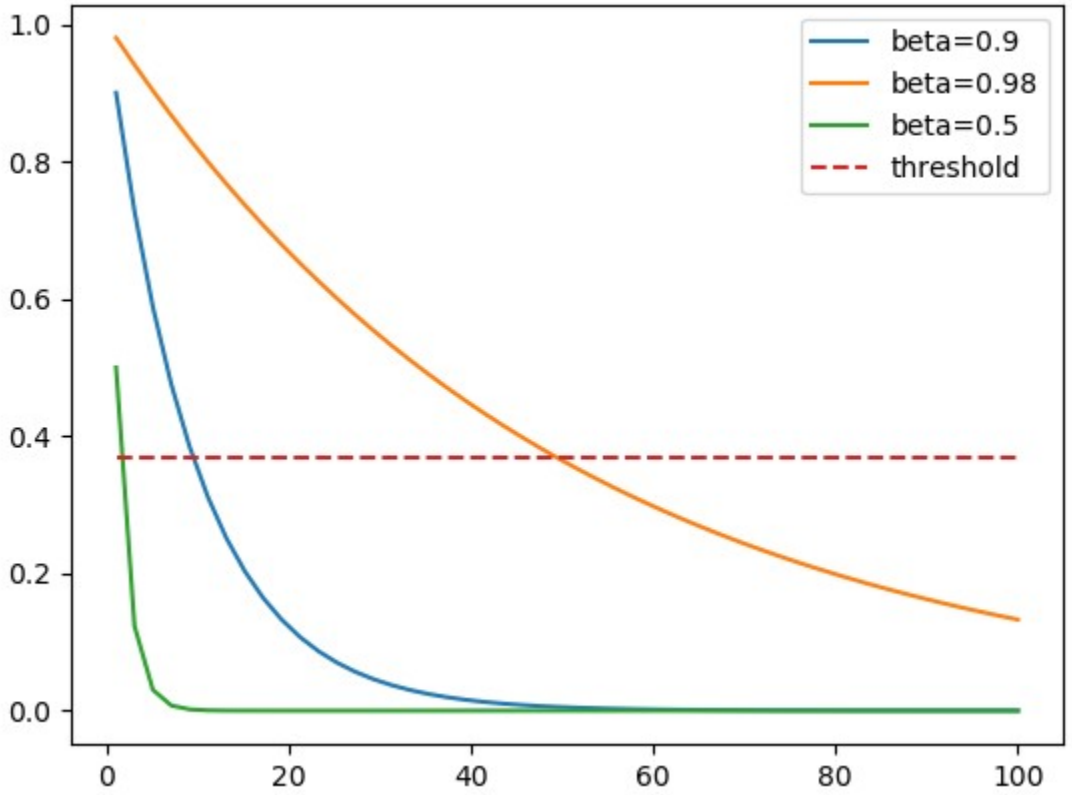
$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$V_{t-1} = \beta V_{t-2} + (1-\beta) S_{t-1}$$
$$V_{t-2} = \beta V_{t-3} + (1-\beta) S_{t-2}$$

$$V_t = \beta(\beta(\beta V_{t-3} + (1-\beta) S_{t-2}) + (1-\beta) S_{t-1}) + (1-\beta) S_t$$

$$V_t = \beta\beta(1-\beta) S_{t-2} + \ldots + \beta(1-\beta) S_{t-1} + \ldots + (1-\beta) S_t$$

The coefficients add up to approximately 1

Exponential weighted average



threshold 1/e

$(1 - \varepsilon)^{1/\varepsilon} = 1/e$

Exponential weighted average

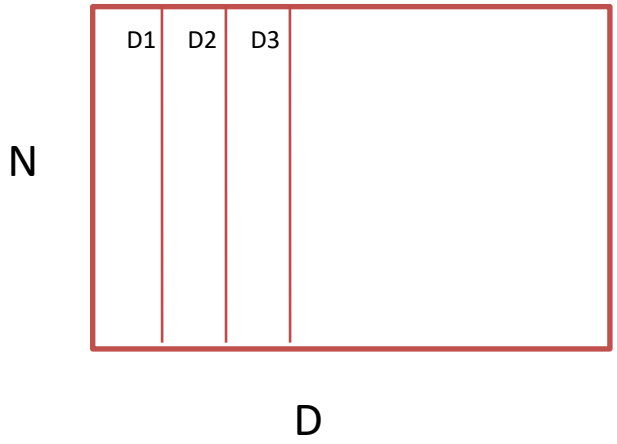
Bias correction

- The first couple of iterations will provide a pretty bad averages because we don't have enough values yet to average over
- Instead of using V_t use the bias corrected version of it:

https://www.youtube.com/watch?v=lAq96T8FkTw
https://www.youtube.com/watch?v=NxTFlzBjS-4
https://www.youtube.com/watch?v=lWzo8CajF5s – bias correction

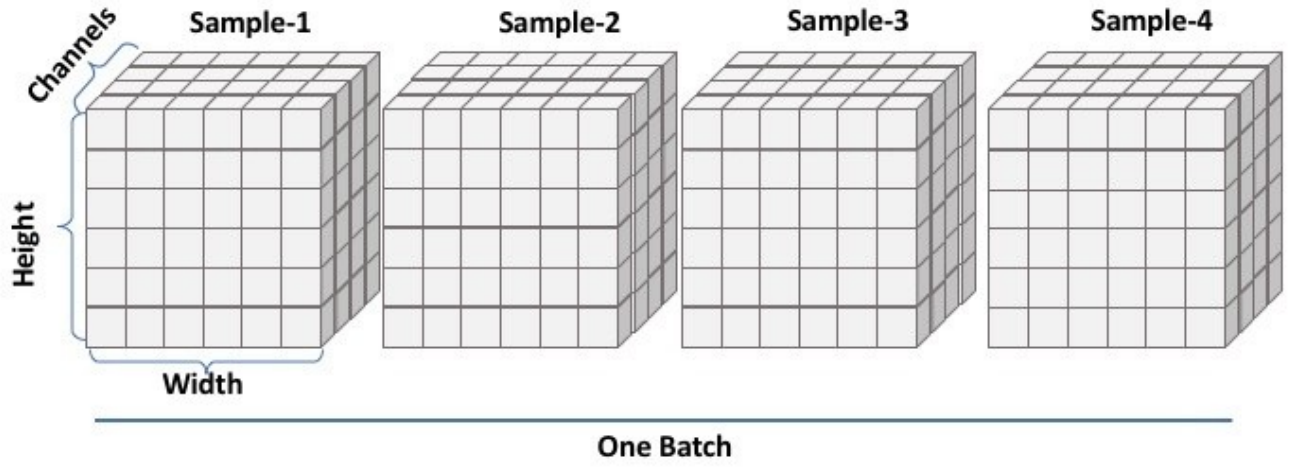
Batch normalization

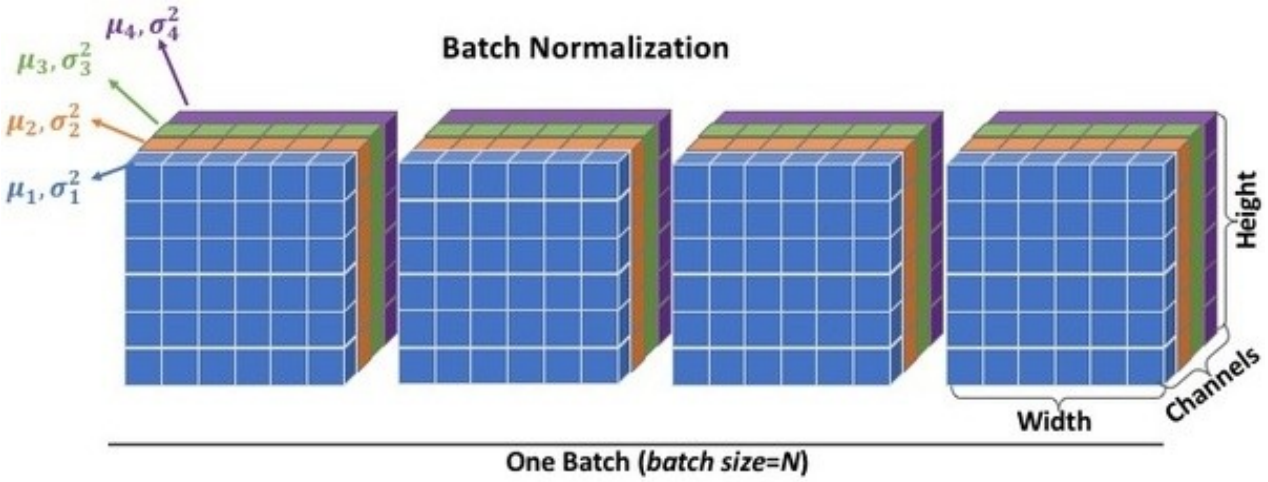
- Compute mean and variance across each dimension



- Normalize

Batch normalization





Batch normalization

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

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

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

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

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

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

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

Simple idea: just make the output of the **linear units** have 0 mean and unit standard deviation

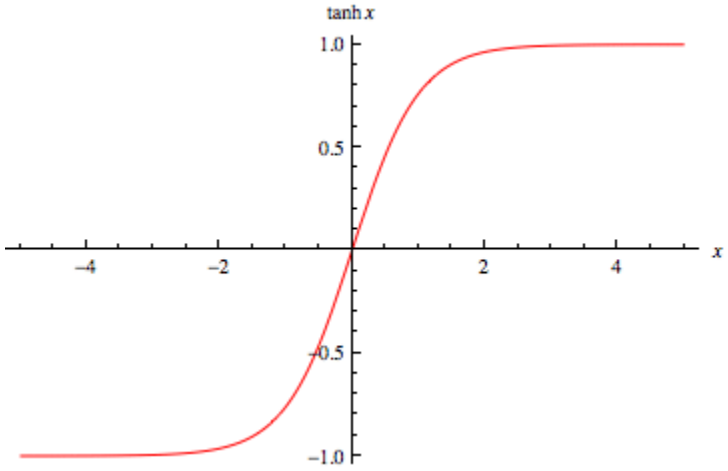
algorithm source: https://arxiv.org/abs/1502.03167

Batch normalization

Normalize:

$$\mu = \frac{1}{m}\sum_i z^{(i)} \qquad \sigma^2 = \frac{1}{m}\sum_i \left(z^{(i)} - \mu\right)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$



Allow the network to modify the range

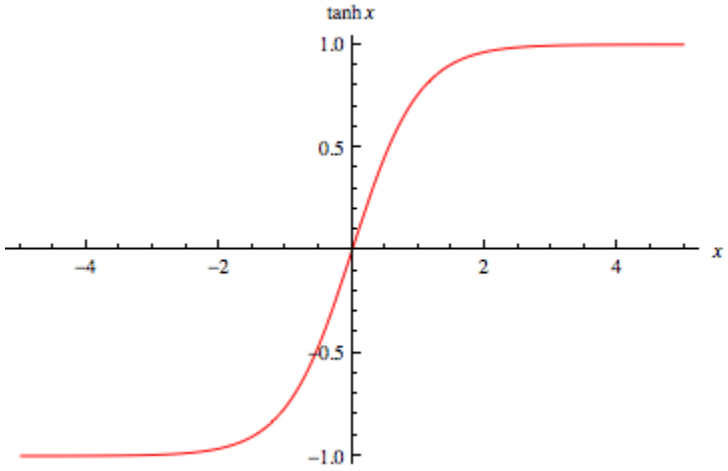
$$\tilde{z}^{(i)} = \gamma\, z^{(i)}_{norm} + \beta$$

Batch normalization

Normalize:

$$\mu = \frac{1}{m}\sum_i z^{(i)} \qquad \sigma^2 = \frac{1}{m}\sum_i \left(z^{(i)} - \mu\right)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$



Allow the network to modify the range

$$\tilde{z}^{(i)} = \gamma\, z^{(i)}_{norm} + \beta$$

Can learn the identity function

$$= \sqrt{\sigma^2}$$

=

Batch normalization at test time

- We cannot compute the mean and standard deviation on a batch of samples
- Instead we use a static mean and standard deviation were computed empirically during training
  - Compute the mean and standard deviation on the entire training set
  - **Estimate the mean and standard deviation using exponential weighted averages**

Batch normalization (advantages)

- Reduces the dependence on weights initialization
- Improves gradient flow through the network
- Allows you to set a higher learning rate
- Slight regularization effect

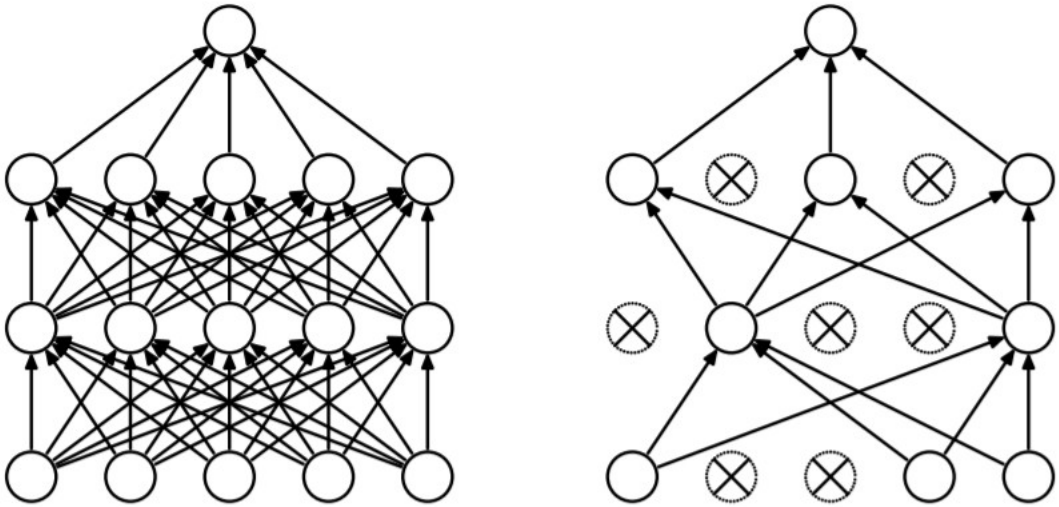
Batch normalization

- Further reading (batch normalization and transfer learning):

https://keras.io/guides/transfer_learning/

Dropout (2014)

Randomly ***drop units*** (and their connections) from the neural network ***during training***



(a) Standard Neural Net

(b) After applying dropout.

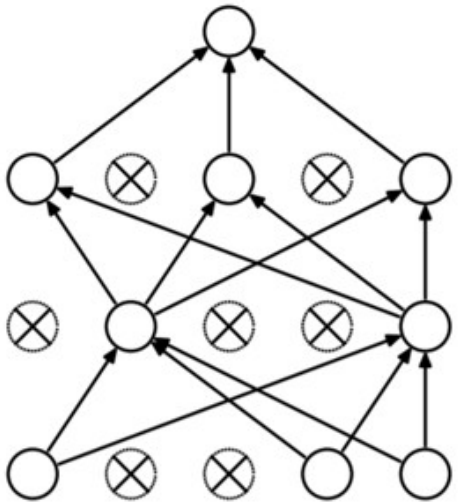
SRIVASTAVA, Nitish, et al. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014, 15.1: 1929-1958.

https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Dropout

keep_prop – probability of keeping a neuron; larger value, less dropout

```
# LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
Z1 = np.dot(W1, X) + b1
A1 = relu(Z1)

# Step 1: initialize matrix D1 = np.random.rand(..., ...)
D1 = np.random.rand(A1.shape[0], A1.shape[1])
# Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
D1 =  (D1 < keep_prob).astype(int)
# Step 3: shut down some neurons of A1
A1 = A1*D1


Z2 = np.dot(W2, A1) + b2
A2 = relu(Z2)

 # Step 1: initialize matrix D2 = np.random.rand(..., ...)
D2 = np.random.rand(A2.shape[0], A2.shape[1])
# Step 2: convert entries of D2 to 0 or 1 (using keep_prob as the threshold)
D2 = (D2 < keep_prob).astype(int)
# Step 3: shut down some neurons of A2
A2 = A2*D2
```



Dropout visualization

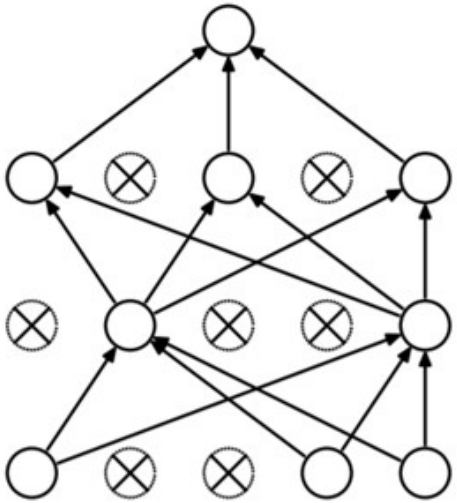
Video source: https://www.coursera.org/learn/deep-neural-network/ungradedLab/xioPP/lab

Dropout intuition

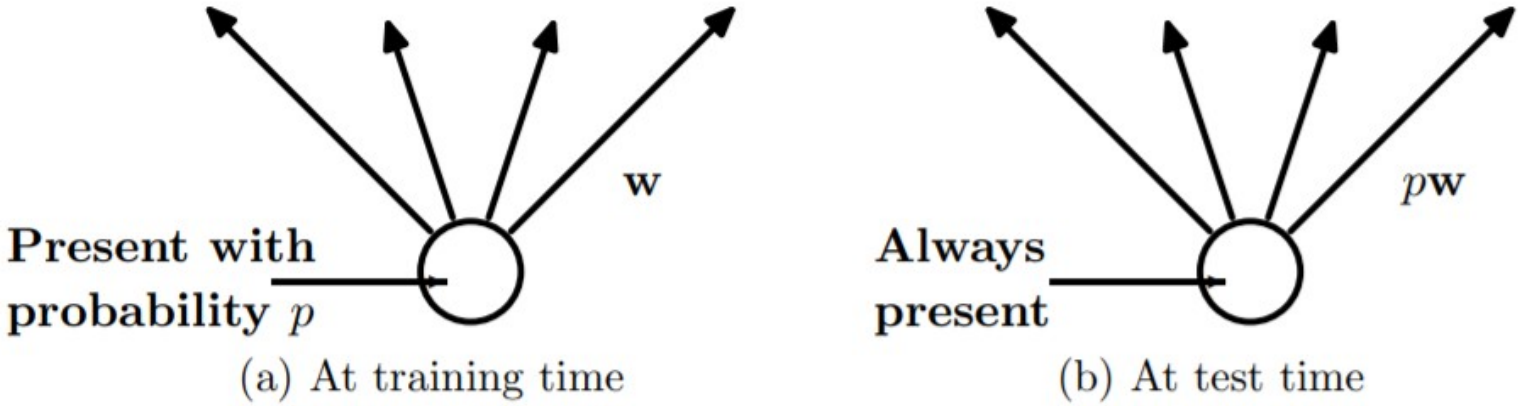
- At each iteration you actually modify your model:
  - train a different model that uses only a subset of your neurons
  - neurons thus become less sensitive to the activation of one other specific neuron (others neuron might be shut down at any time)

Dropout intuition

- Train a large ensemble of models (with shared parameters)
- Forces the model to have a redundant representation



Dropout at test time



(a) At training time (b) At test time

Dropout at test time

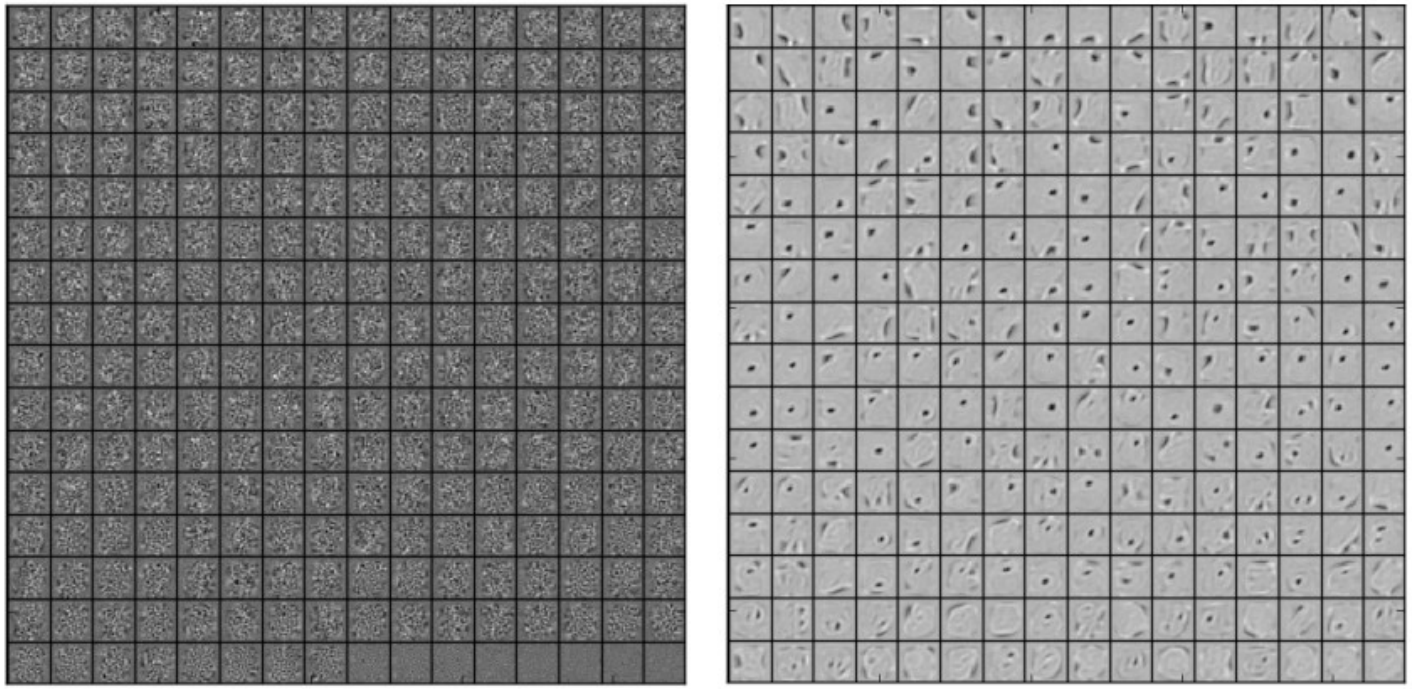
- The neurons are always turned on
- Scale the activation for each neuron
  - output at test time = expected output at training time

Dropout at test time

keep_prop – probability of keeping a neuron; larger value, less dropout

```
# LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
Z1 = np.dot(W1, X) + b1
A1 = relu(Z1)

D1 = np.random.rand(A1.shape[0], A1.shape[1])      # Step 1: initialize matrix D1 = np.random.rand(..., ...)
D1 =  (D1 < keep_prob).astype(int)                 # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
A1 = A1*D1                                         # Step 3: shut down some neurons of A1
A1 = A1/keep_prob                                  # Step 4: scale the value of neurons that haven't been shut down

Z2 = np.dot(W2, A1) + b2
A2 = relu(Z2)

D2 = np.random.rand(A2.shape[0], A2.shape[1])      # Step 1: initialize matrix D2 = np.random.rand(..., ...)
D2 = (D2 < keep_prob).astype(int)                  # Step 2: convert entries of D2 to 0 or 1 (using keep_prob as the threshold)
A2 = A2*D2                                         # Step 3: shut down some neurons of A2
A2 = A2/keep_prob                                # Step 4: scale the value of neurons that haven't been shut down

Z3 = np.dot(W3, A2) + b3
A3 = sigmoid(Z3)
```

Dropout in practice



(a) Without dropout

(b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

A hidden unit cannot rely on other specific units to correct its mistakes. It must perform well in a wide variety of different contexts provided by the other hidden units.

`Dropout` class

```
tf.keras.layers.Dropout(rate, noise_shape=None, seed=None, **kwargs)
```

Data augmentation

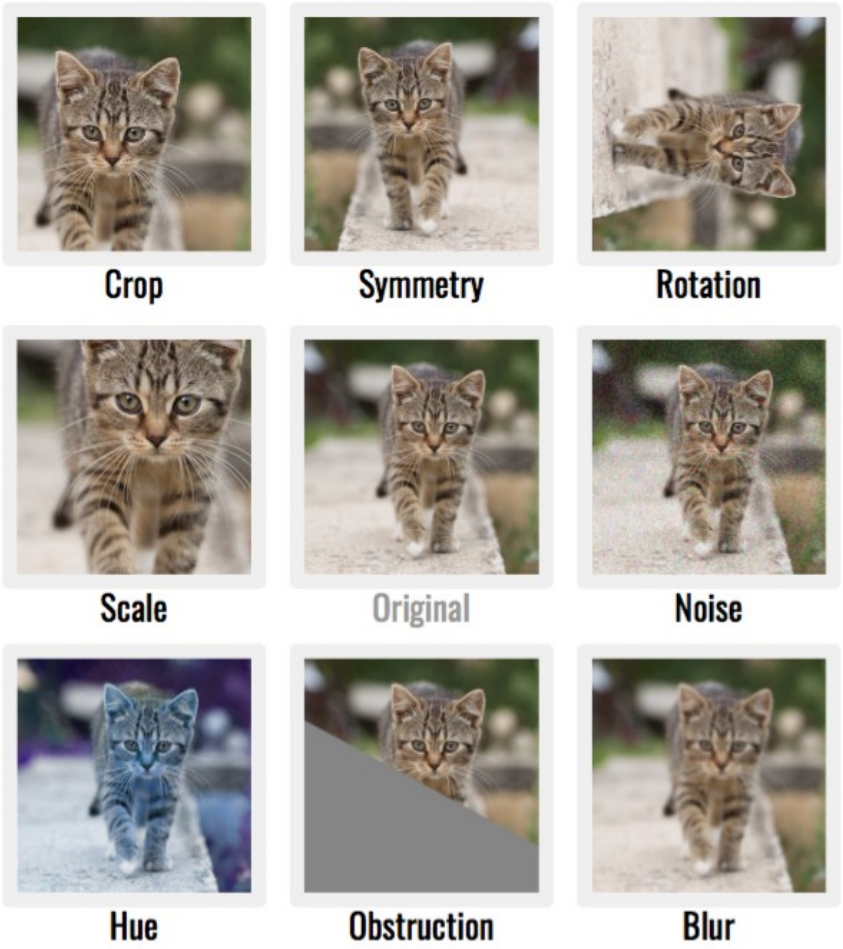
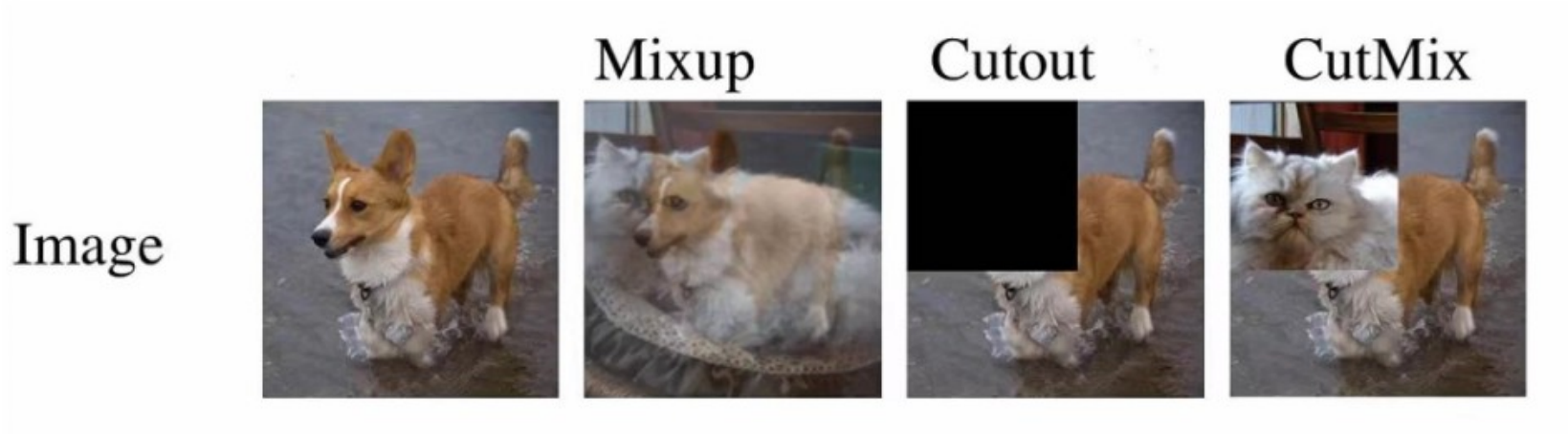


Image source: https://mc.ai/easy-way-to-improve-image-classifier-performancepart-1%E2%80%8A-%E2%80%8Amixup-augmentation-with-codes-2/

Data augmentation



keras – data augmentation

`ImageDataGenerator` class

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=1e-06,
    rotation_range=0,
    width_shift_range=0.0,
    height_shift_range=0.0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.0,
    channel_shift_range=0.0,
    fill_mode="nearest",
    cval=0.0,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    data_format=None,
    validation_split=0.0,
    dtype=None,
)
```

Training a neural network

Optimization algorithms

Parameters update – gradient descent

Vanilla gradient descent:

1. Forward propagation through the network
2. Compute loss
3. Back-propagate to compute gradients
4. Update the parameters using the gradient

Parameters update – gradient descent

1. Batch gradient descent
   - "classical" implementation of gradient descent
   - Use vectorization and process the entire training set at the same time
2. Split the training set into *mini-batches*

Parameters update – gradient descent

…. ….]

…. ….]

Minibatch *t*:

Example *i:*

Parameters update – gradient descent

- "Classical" gradient descent
  - Use vectorization and process the entire training set at the same time

- Split the training set into *mini-batches*

```
while True:
    batch = sample_batch(data)
    W_gradient = compute_gradient(loss_func, batch,
W)
    # update parameters
    W += -lr*W_gradient
```

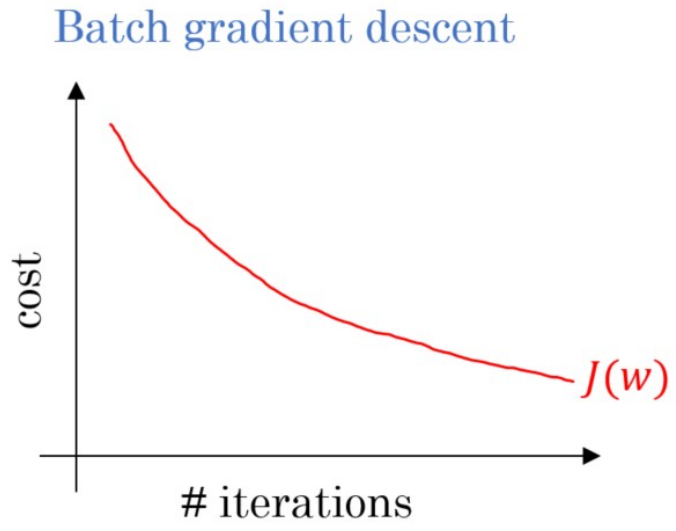
Mini-batch gradient descent

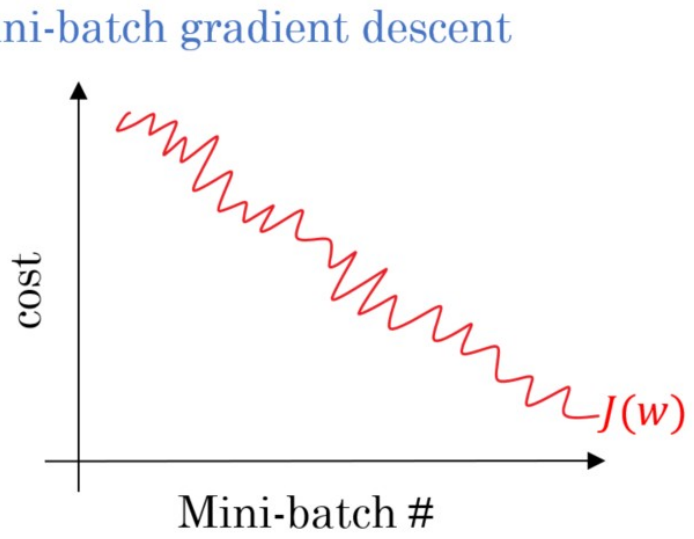
- *Epoch*: single pass through the entire training set

```
270/270 [==============================] - 10s 37ms/step - loss: 0.0451 - mae_metric: 32.8503
Epoch 37/400
270/270 [==============================] - 10s 37ms/step - loss: 0.0555 - mae_metric: 37.9888
Epoch 38/400
270/270 [==============================] - 10s 37ms/step - loss: 0.0557 - mae_metric: 39.1687
Epoch 39/400
270/270 [==============================] - 10s 37ms/step - loss: 0.0471 - mae_metric: 31.8123
Epoch 40/400
270/270 [==============================] - 10s 36ms/step - loss: 0.0293 - mae_metric: 32.6505
Epoch 41/400
270/270 [==============================] - 10s 37ms/step - loss: 0.0412 - mae_metric: 32.2138
Epoch 42/400
270/270 [==============================] - 10s 37ms/step - loss: 0.0601 - mae_metric: 37.9935
Epoch 43/400
220/270 [=======================>......] - ETA: 1s - loss: 0.0550 - mae_metric: 33.8031
```

Size of the mini-batches

- Stochastic gradient descent (m=1)
- Mini-batch gradient descent
- Batch gradient descent (m = M)





Size of the mini-batches

- Stochastic gradient descent (m=1)
  - no vectorization, lose speedup
- Mini-batch gradient descent
  - Fastest learning: use vectorization and doesn't take top much time to update the weights
  - 2^6 , 2^7 , …, 2^{10}
- Batch gradient descent (m = M)
  - Too much time per iteration

Parameters update – mini-batch gradient descent

foreach batch *t:*

1. Forward propagation through the network
2. Compute loss
3. Backpropagate to compute gradients
4. Update the parameters using the gradient