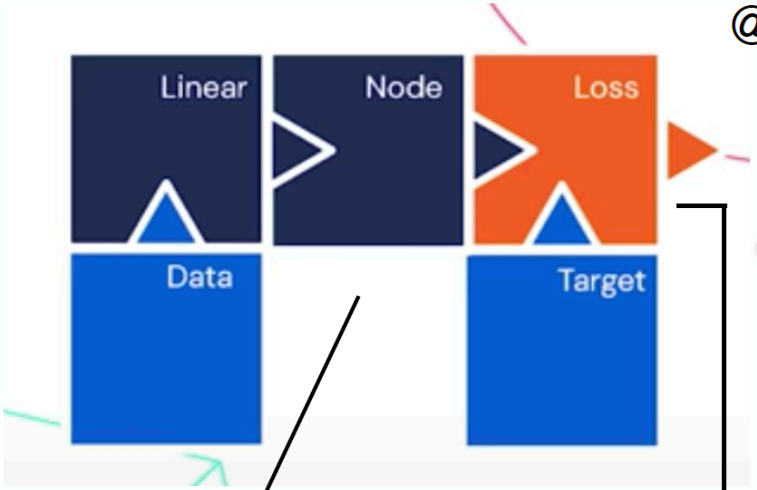
RECAP:

(deepmind @Czarnecki)

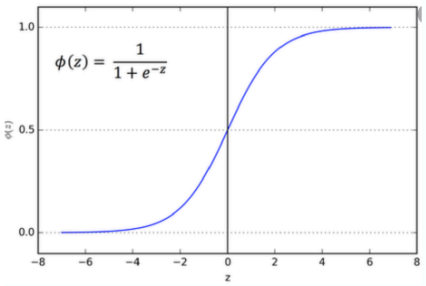


$$f_W(\vec{x}) = \vec{y}$$

$$\vdots$$

$$g_3(W_3 g_2(W_2 g_1(W_1 \vec{x_0})))$$

activation function



$$\frac{1}{N}\sum_{i=1}^{N}(y_i - f_w(x))^2$$

Minimized through gradient descent (backpropagation)

Neural Networks as statistical models

$$f_W(\vec{x}) = \vec{y} \quad \Leftrightarrow \quad p(y|x;W)$$

we have a set of independent realizations:

$$(x_i, y_i); i = 1, ..., N$$

and want to find the underlying probability distribution of y given x - p(y|x):

If we assume that p(y|x) is a Gaussian distribution (from Zejko's lecture yesterday):

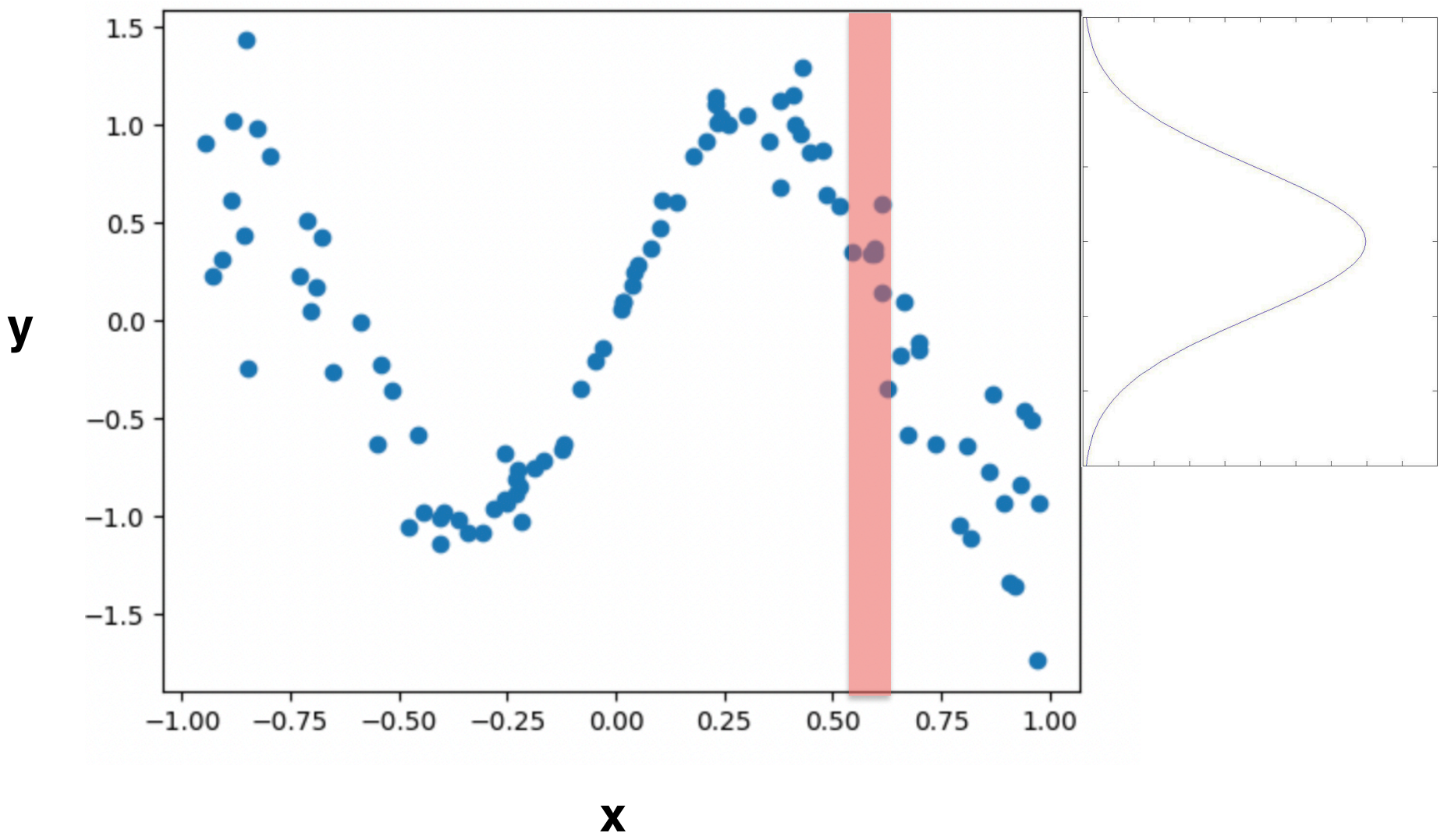
$$p_w(y|x) = L(w, \sigma^2) = \prod_{i=1}^{N} (2\pi\sigma^2)^{-N/2} exp(-\frac{1}{\sigma^2}(y_i - f_w(x_i))^2)$$

Network output

weights

So the goal is to find the values of w (weights) that estimate the mean of a Gaussian distribution for y given a set of N independent observations

$$p(y|x) = \mathcal{N}(\mu, \sigma)$$



we have a set of independent realizations:

$$(x_i, y_i); i = 1, ..., N$$

and want to find the underlying probability distribution of y given x [p(y|x)]:

Since y can only take 0 / 1 values, we assume it can be parametrized with a Bernouilli distribution:

$$P(y_i = 0|x_i) = 1 - sigm(f_w(x_i)) \qquad P(y_i = 1|x_i) = sigm(f_w(x_i))$$

sigmoid function

weights

So the goal is to find the values of w (weights) that generate a Bernouilli distribution for y given a set of N independent observations

Regressions

$$p_w(y|x) = L(w, \sigma^2) = \prod_{i=1}^{N} (2\pi\sigma^2)^{-N/2} exp(-\frac{1}{\sigma^2}(y_i - f_w(x_i))^2)$$

weights (model parameter)

neural network

And then take the log:

$$ln\ L = cst - \sum_{i=1}^{N} \frac{(x_i - f_w(x_i))^2}{2\sigma^2}$$

And now assume sigma = 1:

$$ln\ L = -\sum_{i=1}^{N} (x_i - f_w(x_i))^2 = MSE$$

Regressions

WHEN USING THE MEAN SQUARED ERROR, WE ARE ASSUMING THAT THE POSTERIOR P(y|x) IS A NORMAL DISTRIBUTION.

OUR ESTIMATOR IS THEREFORE THE MEAN OF THE NORMAL PDF WITH SIGMA = 1 THAT MAXIMIZES THE LIKELIHOOD (MAXIMUM LIKELIHOOD ESTIMATION).

What about classification?

$$H = -\frac{1}{N}\sum_{i=1}^{N} y_i.log(f_w(x_i)) + (1 - y_i).log(1 - f_w(x_i))$$

GROUND TRUTH

GROUND TRUTH

NETWORK OUTPUT

NETWORK OUTPUT

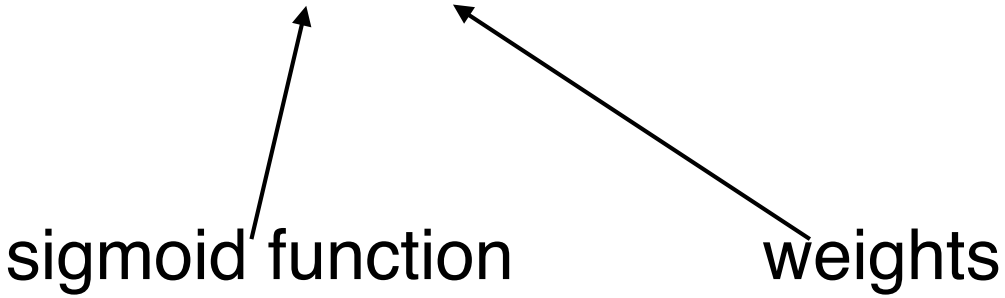
we have a set of independent realizations:

$$(x_i, y_i); i = 1, ..., N$$

and want to find the underlying probability distribution of y given x:

Since y can only take 0 / 1 values, we assume it can be parametrized with a **Bernouilli distribution**:

$$P(y_i = 0|x_i) = 1 - sigm(f_w(x_i)) \qquad P(y_i = 1|x_i) = sigm(f_w(x_i))$$



So the goal is to find the values of w (weights) that generate a Bernouilli distribution for y given a set of N independent observations

This can be achieved via Maximum Likelihood estimation:

The likelihood of a given observation under the Bernouilli assumption can be written as:

$$L(w; x_i, y_i) = [sigm(f_w(x_i))]^{y_i} [1 - sigm(f_w(x_i))]^{1-y_i}$$

which is equal to $[sigm(f_w(x_i))]$ if y=1 and $[1 - sigm(f_w(x_i))]$ if y=0

And the likelihood of the entire sample is the product of the likelihoods:

$$L(w; x_i, y_i) = \prod_{i=1}^{N} [sigm(f_w(x_i))]^{y_i} [1 - sigm(f_w(x_i))]^{1-y_i}$$

So, the log-likelihood:

$$l(w; x_i, y_i) = \sum_{i=1}^{N} y_i \times \log[sigm(f_w(x_i))] + (1 - y_i) \times \log[1 - sigm(f_w(x_i))]$$

THEREFORE EQUIVALENT TO THE CROSS-ENTROPY LOSS:

$$l(w; x_i, y_i) = \sum_{i=1}^{N} y_i \times \log[sigm(f_w(x_i))] + (1 - y_i) \times \log[1 - sigm(f_w(x_i))]$$

$$H = -\frac{1}{N} \sum_{i=1}^{N} y_i.log(f_w(x_i)) + (1 - y_i).log(1 - f_w(x_i))$$

WHEN YOU DO A BINARY CLASSIFICATION WITH NEURAL NETWORKS YOU ARE SIMPLY FINDING THE WEIGHTS OF YOUR MODEL THAT MAXIMIZE THE LIKELIHOOD OF A BERNOUILLI DISTRIBUTION

ASSUMES THAT THE POSTERIOR p(ylx) CAN BE MODELLED BY A BERNOUILLI DISTRIBUTION

For regressions: let's do some **"standard" bayesian inference**

goal of inference

posterior ***likelihood*** ***prior***

$$p(\theta \mid X) = \frac{p(X \mid \theta)\, p(\theta)}{p(X)}$$

$$\ln p(X \mid \theta) = \ln \mathscr{L} = \left(X - m(\theta)\right)^{T} \mathbf{C}^{-1} \left(X - m(\theta)\right)$$

Mean of gaussians

model

covariance matrix

For regressions: let's do some **"standard" bayesian inference**

goal of inference

posterior *likelihood* *prior*

$$p(\theta \mid X) = \frac{p(X \mid \theta)\, p(\theta)}{p(X)}$$

$$\ln p(X \mid \theta) = \ln \mathscr{L} = \left(X - m(\theta)\right)^T \mathbf{C}^{-1} \left(X - m(\theta)\right)$$

Mean of gaussians

model

covariance matrix

If all Gaussians are assumed to be independent and with standard deviation =1, then:

$$\mathbf{C}^{-1} = \mathbf{I} \qquad ln\ p(X|\theta) = (X - m(\theta))^2 = MSE$$

$$ln\ p(X|\theta) = (X - m(\theta))^2 = MSE$$

If you do not care about the posterior, one way to find the parameters of the model theta, is by maximising the likelihood: **maximum likelihood estimation, i.e. minimising the MSE**

When training a NN with an MSE loss, we are finding the parameters of the network w that maximise the likelihood, under a Gaussian assumption.

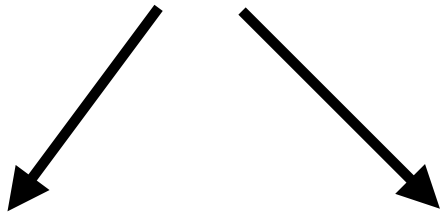
$$ln\ p(X|w) \sim (X - f_w(X)))^2 = MSE$$

Weights of NN

A standard NN performs Maximum Likelihood Estimation of the weights W

$$D = (x^{(i)}, y^{(i)})$$

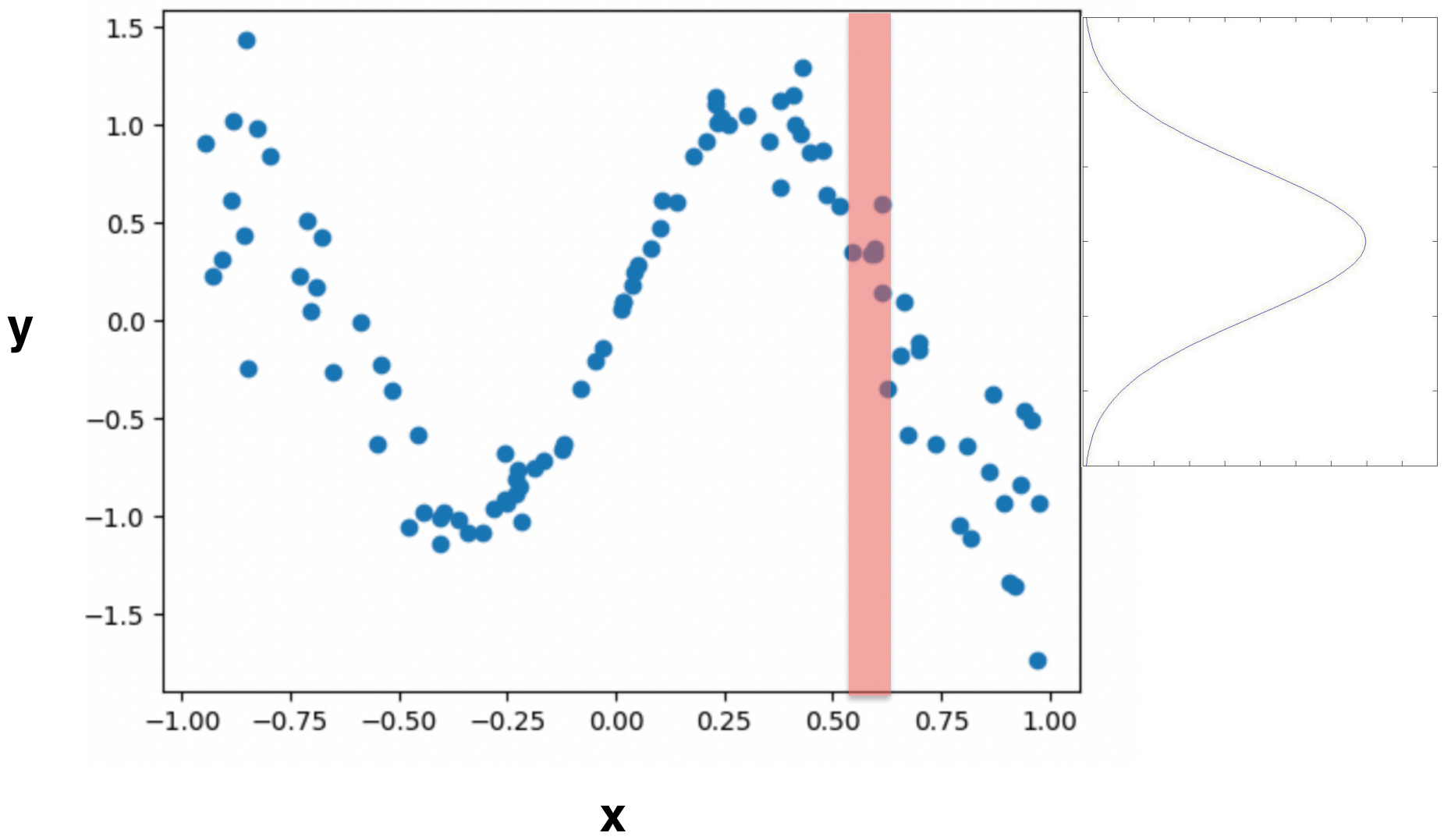
$$p(D|w) = \prod_i p(y^{(i)}|x^{(i)}, W) \sim p_w(y|x)$$



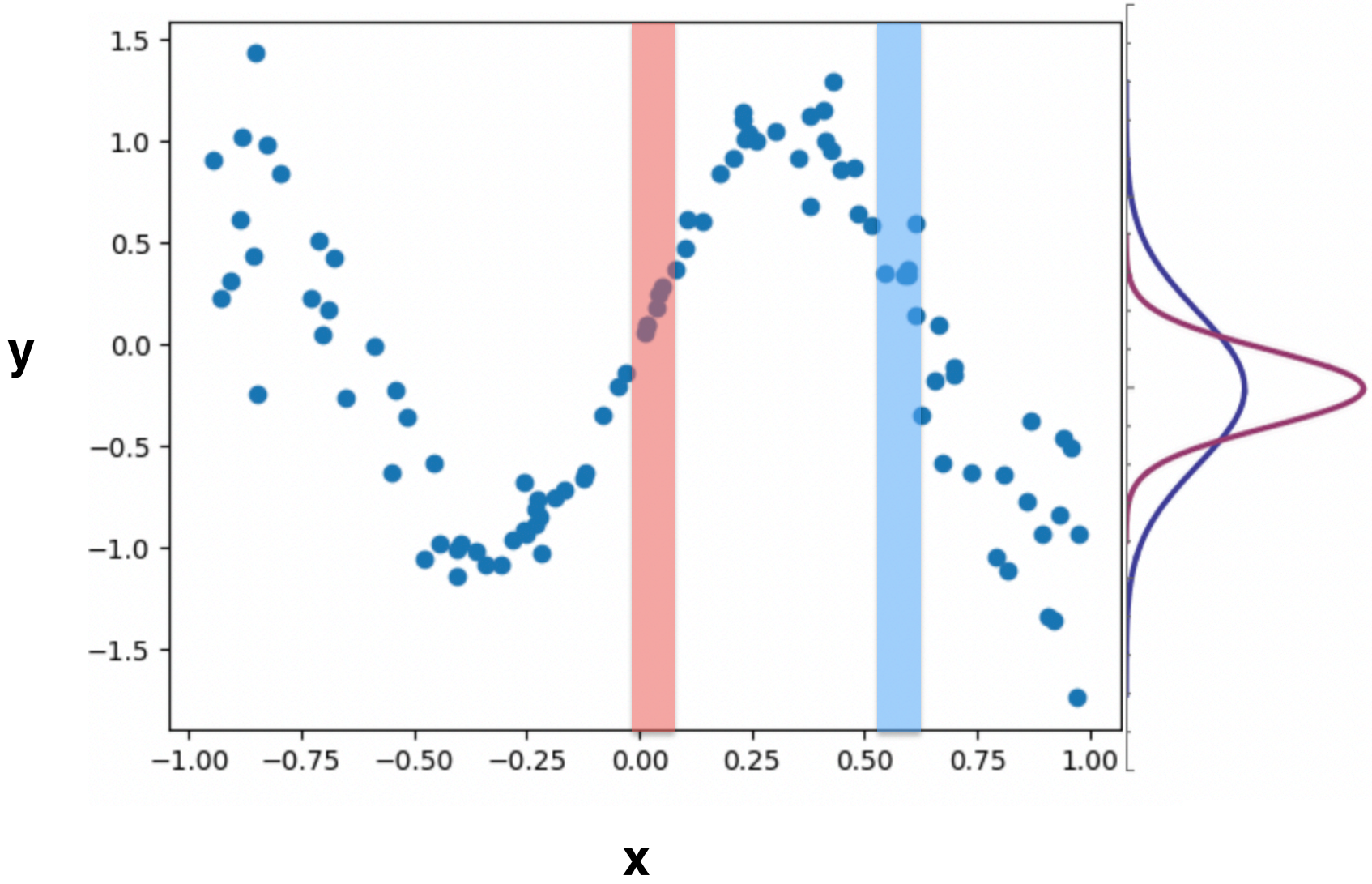
Crossentropy
(~ Bernouilli)

MSE
(~Normal(mu,1))

$$p(y|x) = \mathcal{N}(\mu, 1)$$



$$p(y|x) = \mathcal{N}(\mu, 1) \quad ?$$

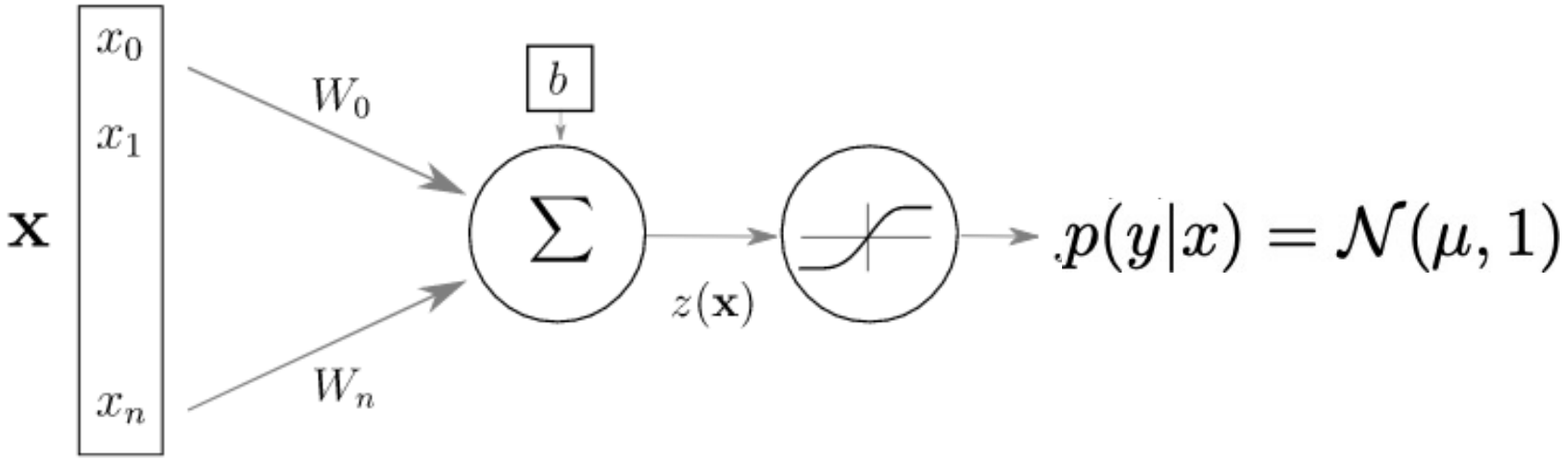


TWO TYPES OF UNCERTAINTY:

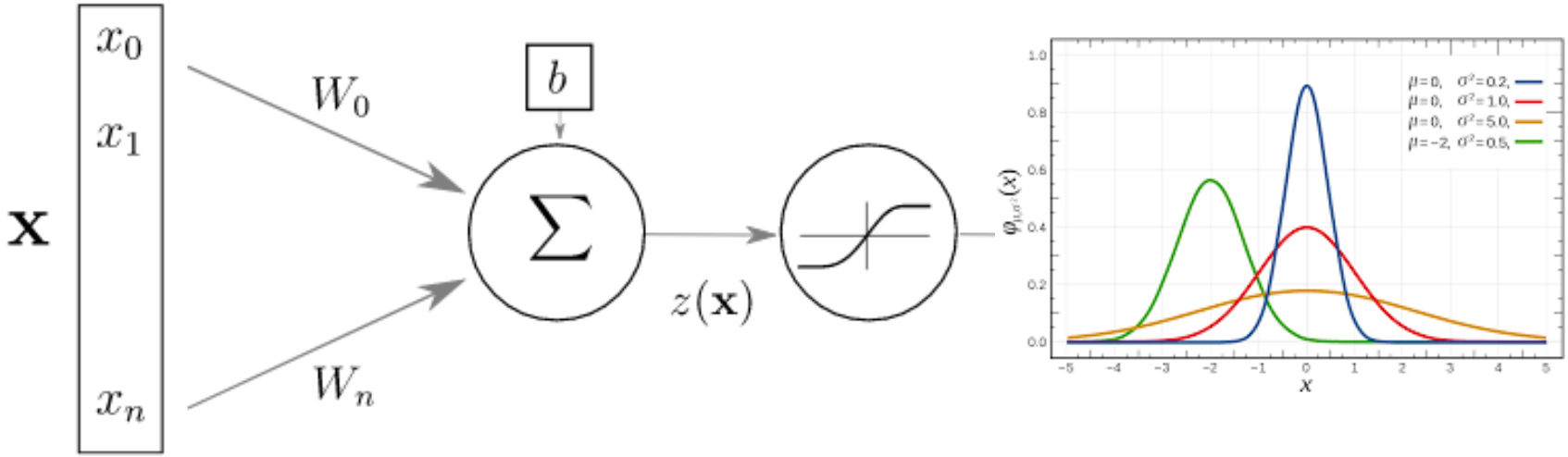
- **Aleatoric (Random)**:Uncertainty coming from the inherent noise in training data. It cannot be reduced if we get more data. Aleatoric uncertainty is covered by the probability distribution used to define the likelihood function

- **Epistemic (Systematics)**: This kind of uncertainty can be reduced if we get more data. Consequently, epistemic uncertainty is higher in regions of no or little training data and lower in regions of more training data

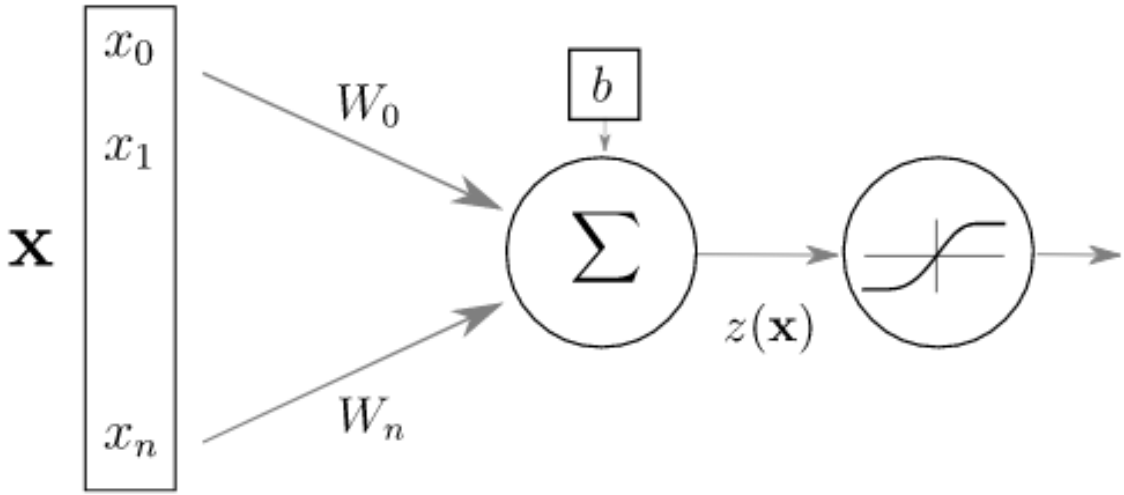
Mixture Density Network



$$p(y|x) = \mathcal{N}(\mu, \sigma)$$

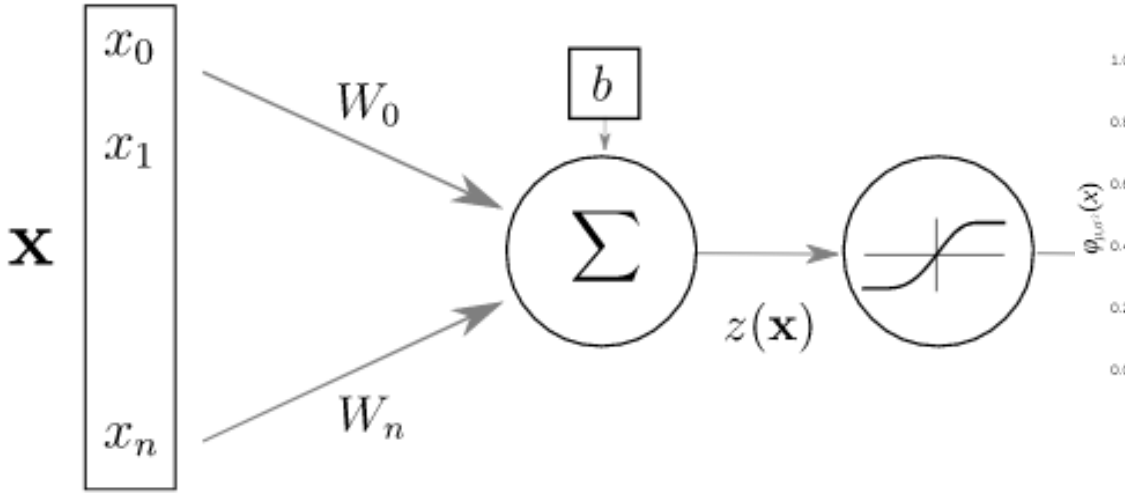


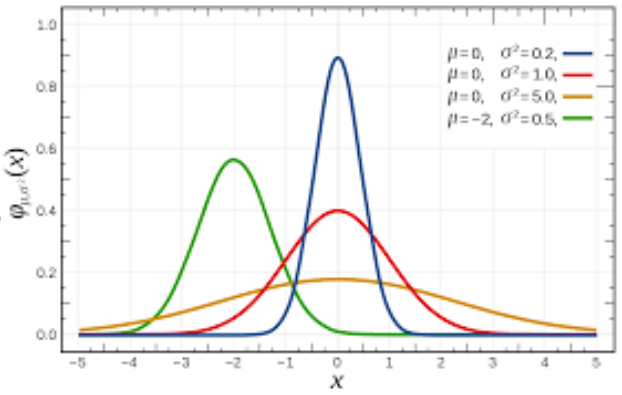
Mixture Density Network



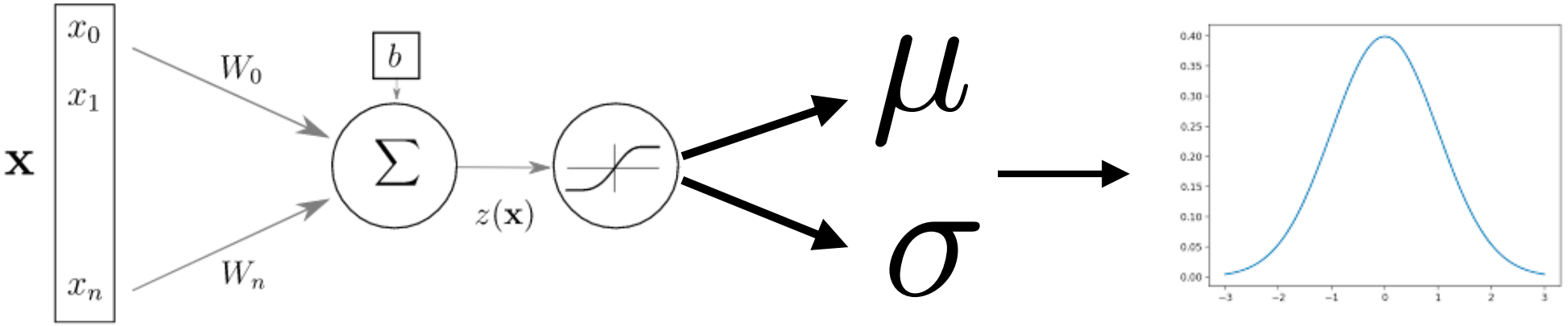
MSE = Max(p(D|w))

$p(D|w)$





Mixture Density Network

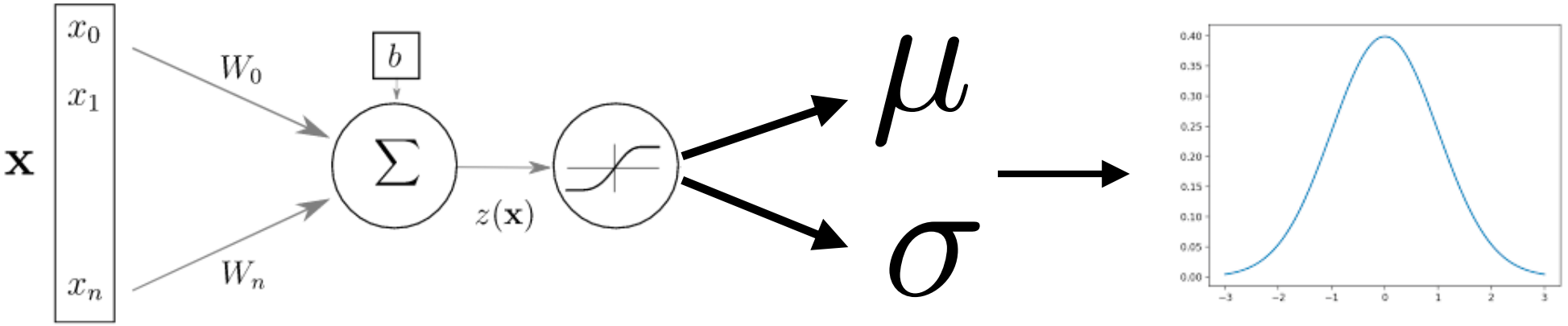


$p(D|w)$

[Likelihood of y under fw(x)]

Reparametrizarion trick: The neural network outputs the parameters of a distribution, on which one can back propagate.

Probabilistic Neural Network

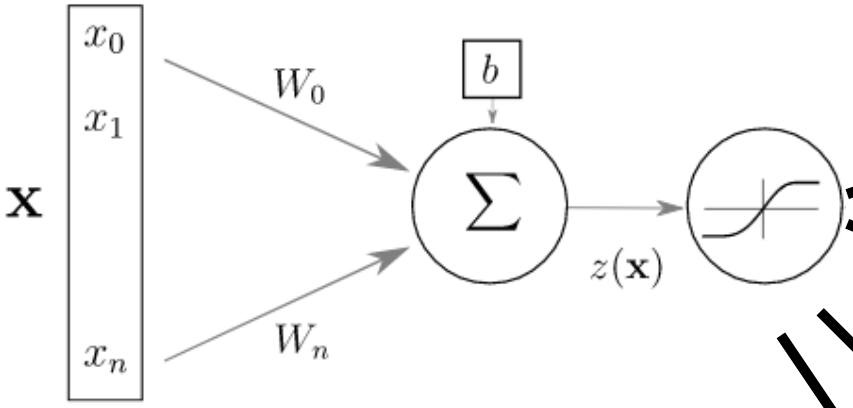


$p(D|w)$

[Likelihood of y under fw(x)]

The loss function becomes then the max of the (log) likelihood

What if the posterior is not gaussian?

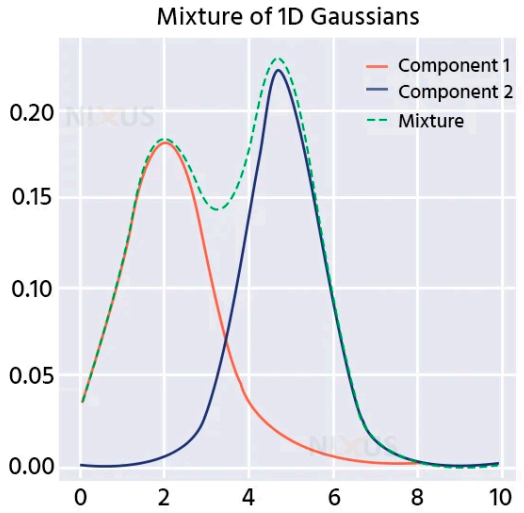


μ

σ

μ

σ



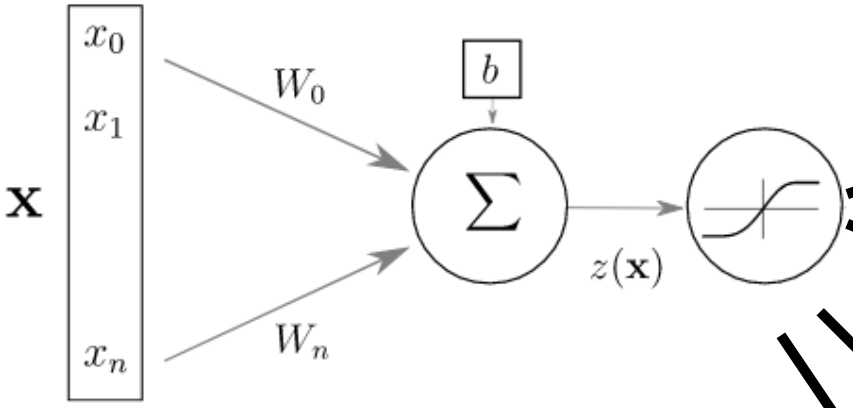
Assume any distribution than
can be parametrized with some parameters and add outputs to the NN

$p(D|w)$

[Likelihood of y under fw(x)]

The loss function remains the max of the (log) likelihood

What if the posterior is not gaussian?

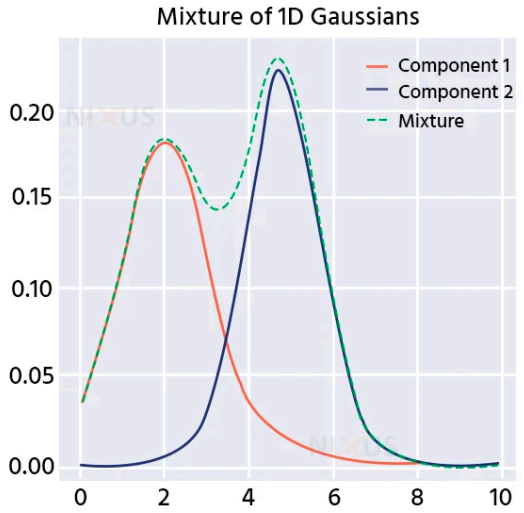


μ

σ

μ

σ



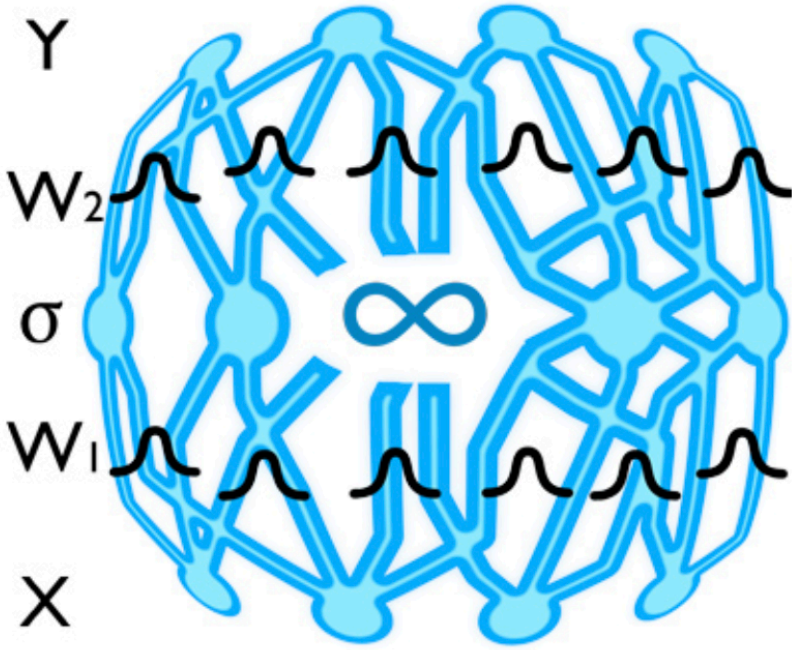
Assume any distribution than
can be parametrized with some parameters and add outputs to the NN

$p(D|w)$

[Likelihood of y under fw(x)]

*Neural Flows allow to estimate pdfs without assuming any parametrization (last lectures)

CAPTURING "MODEL UNCERTAINTY", OR EPISTEMIC UNCERTAINTY, OR UNCERTAINTY ON THE WEIGHTS

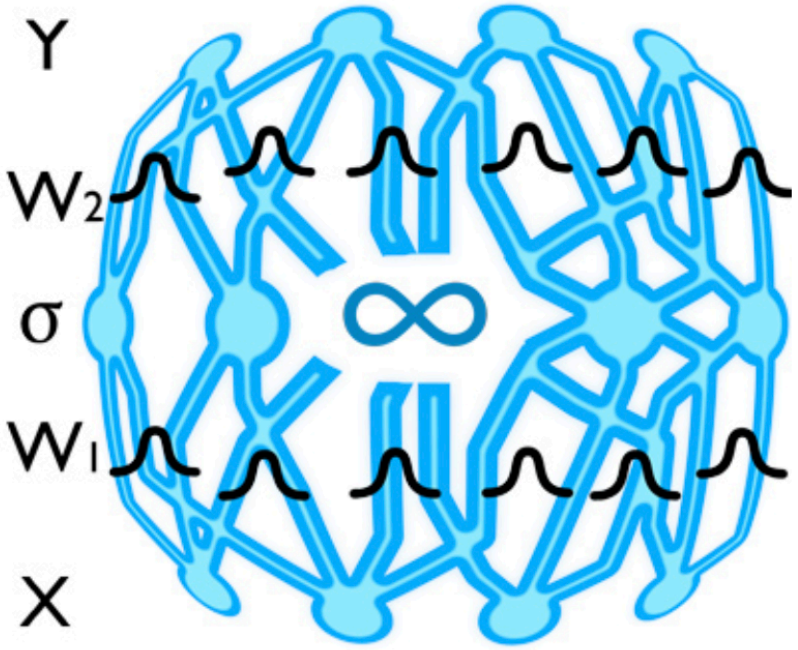


In the previous approach, we are choosing one model among all possible models

IDEALLY, WE WOULD LIKE TO MARGINALIZE OVER ALL MODELS AND OVER ALL POSSIBLE WEIGHT VALUES

$$p(y|x, D) = \int p(y|x, W).p(W|D)dW$$

CAPTURING "MODEL UNCERTAINTY", OR EPISTEMIC UNCERTAINTY



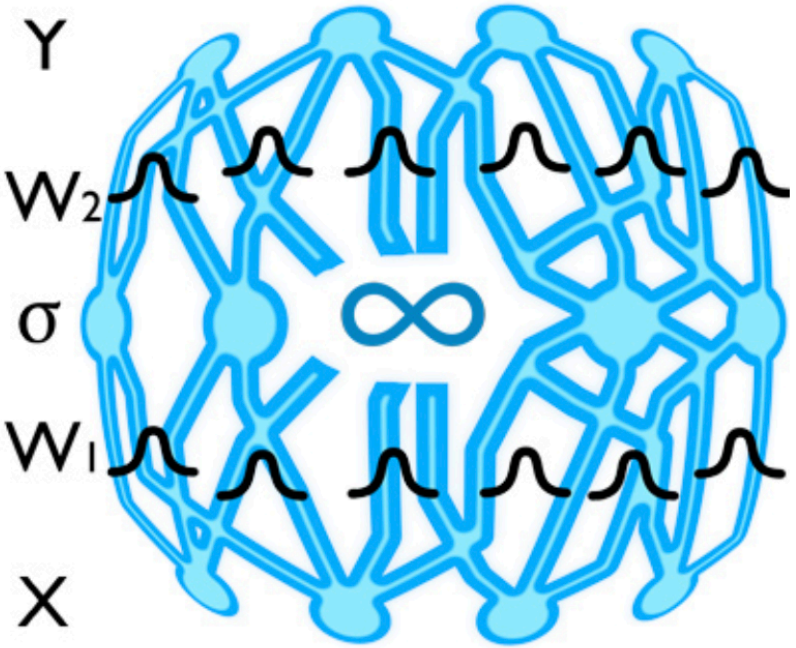
In the previous approach, we are choosing one model among all possible models

IDEALLY, WE WOULD LIKE TO MARGINALIZE OVER ALL MODELS AND OVER ALL POSSIBLE WEIGHT VALUES

$$p(y|x, D) = \int p(y|x, W).p(W|D)dW$$

This is typically intractable

Bayesian Neural Networks are an approach for epistemic uncertainty estimation



BNNs ADD A PRIOR DISTRIBUTION TO EACH WEIGHT - <u>HARD TO TRAIN</u>

Deep Ensembles

$$p(y|x) = M^{-1} \sum_{m=1}^{M} p_{w_m}(y|x, w_m)$$

For classification, this corresponds to averaging the predicted probabilities

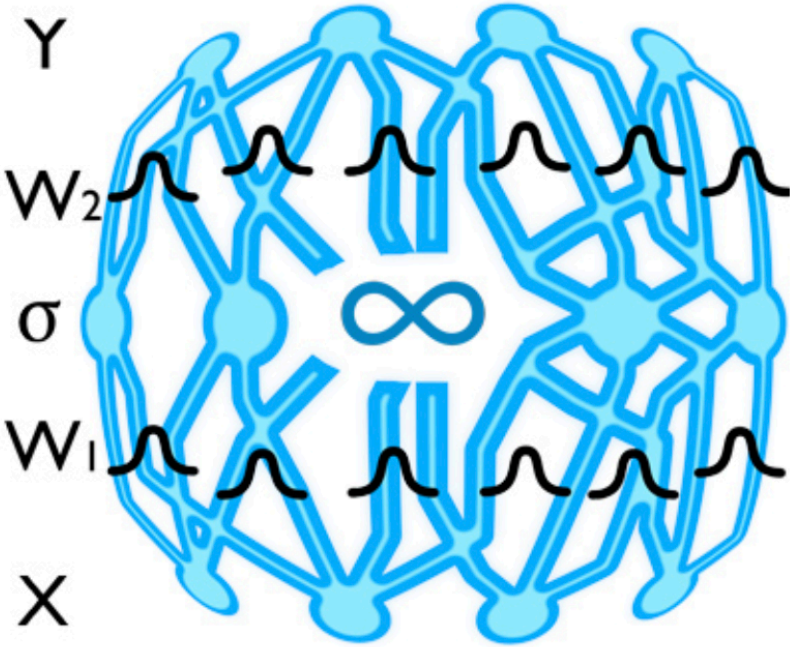
For regression, the prediction is a Mixture of Gaussians.
(One Gaussian / model)

Which can be described with the following summary statistics

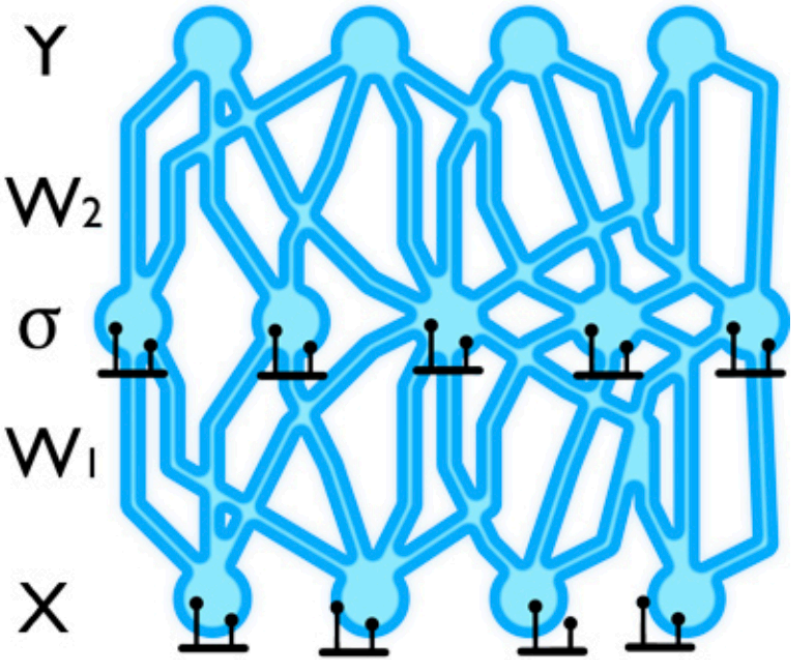
$$\mu_* = M^{-1} \sum_m \mu_{w_m} \qquad \sigma_*^2 = M^{-1} \sum_m (\sigma_{w_m}^2 + \mu_{w_m}^2) - \mu_*^2$$

DROPOUT AS EPISTEMIC UNCERTAINTY ESTIMATION

Denker&LEcun91, Neal+95, Graves+11, Kingma+15, **Gal+15…**



BNNs ADD A PRIOR DISTRIBUTION TO EACH WEIGHT - HARD TO TRAIN



GAL+15 SHOW THAT DROPOUT CAN BE USED TO ESTIMATE UNCERTAINTY

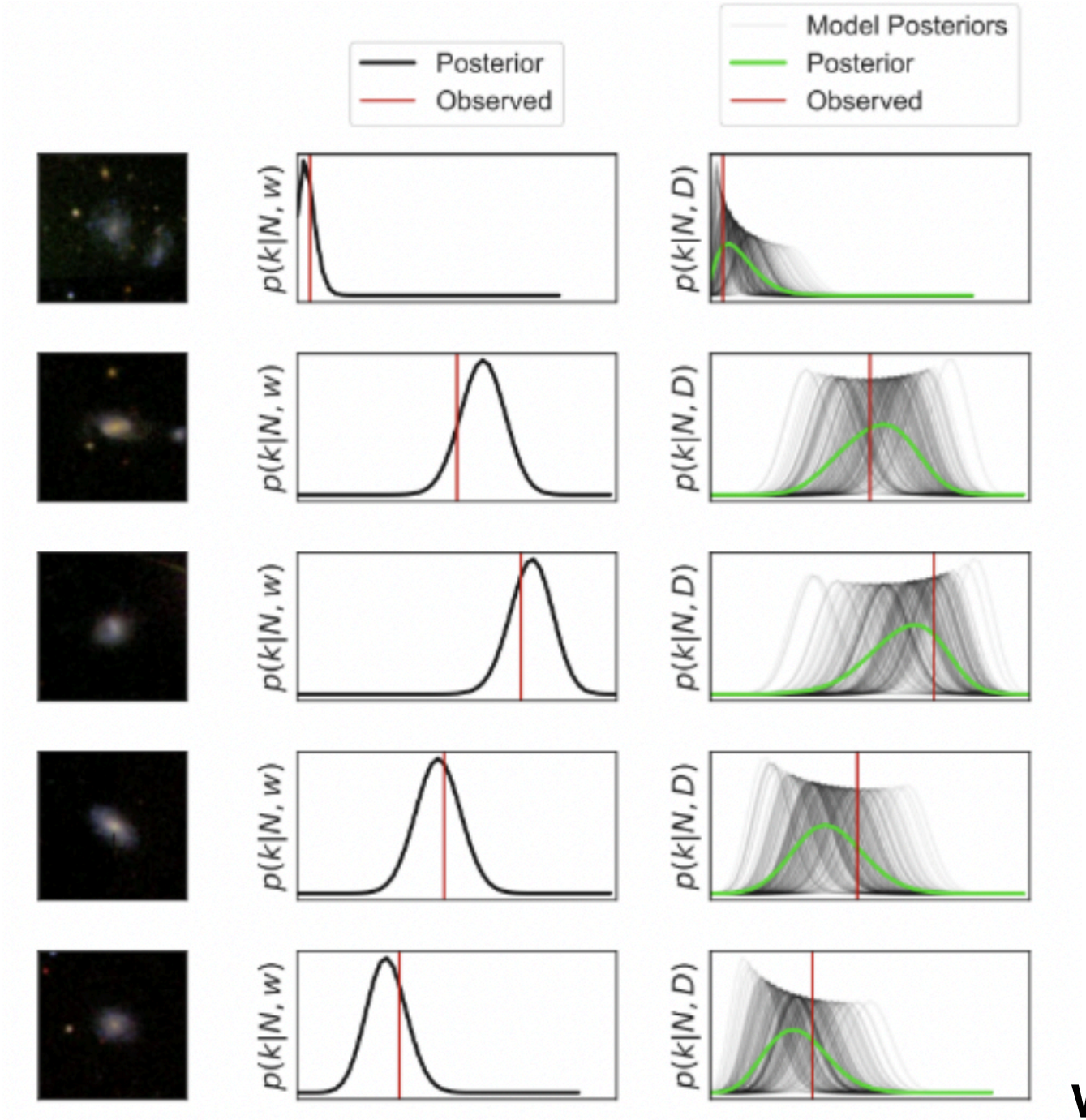
DROPOUT AS EPISTEMIC UNCERTAINTY ESTIMATION

We can treat the many different networks (with different neurons dropped out) as Monte Carlo samples from the space of all available models. This provides mathematical grounds to reason about the model's uncertainty and, as it turns out, often improves its performance.

We simply apply dropout at test time, that's all! Then, instead of one prediction, we get many, one by each model. We can then average them or analyze their distributions.

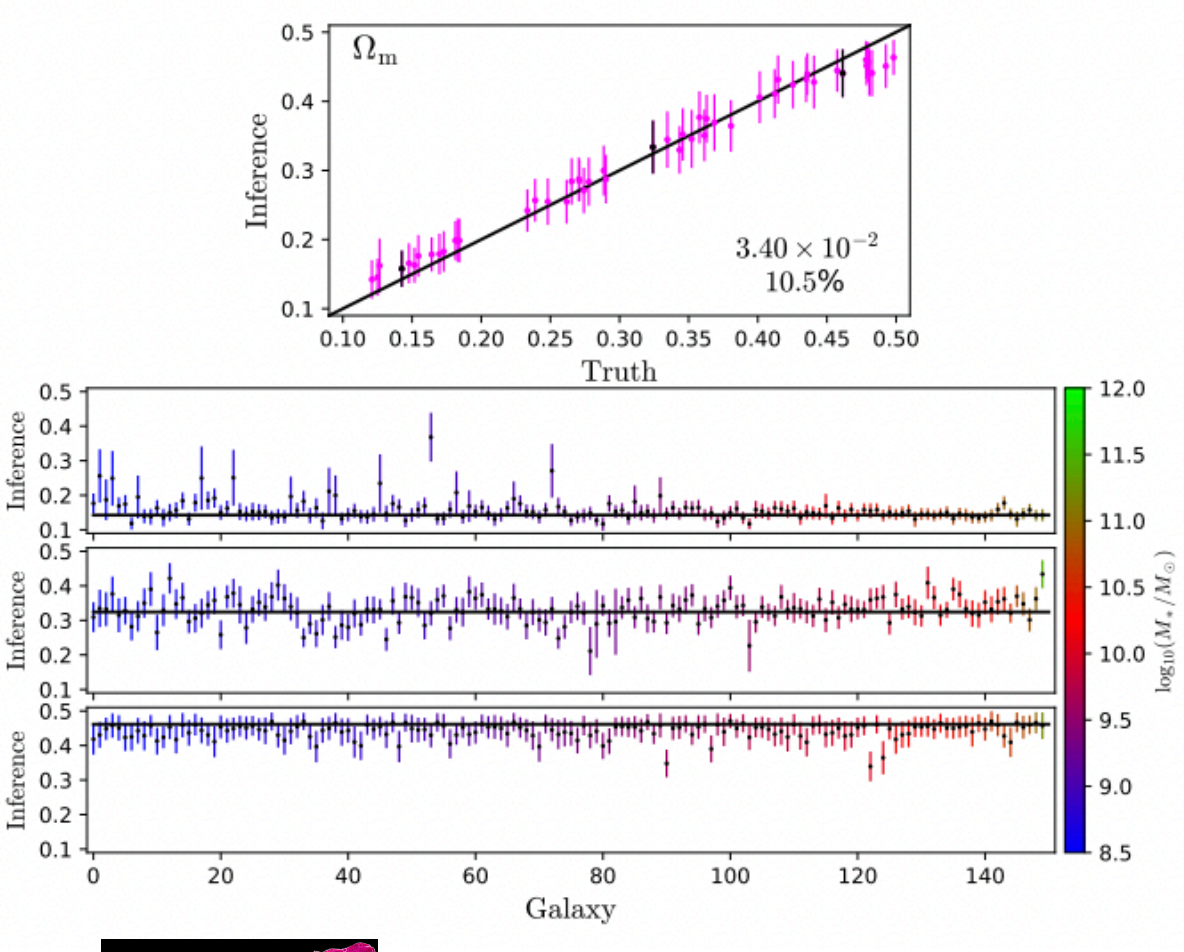
Aleatoric

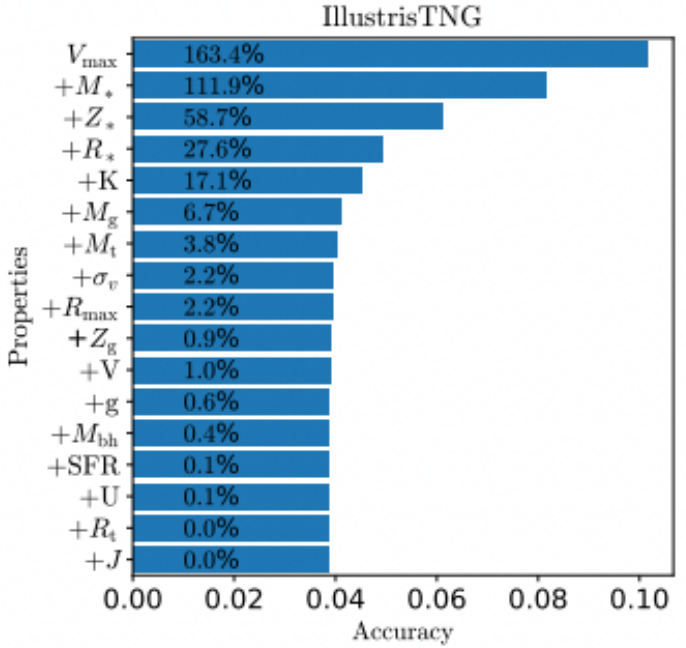
Aleatoric+Epistemic

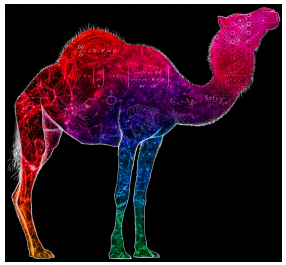


Walmsley+20

Cosmological Inference: beyond summary statistics







(Cosmology and Astrophysics with Machine Learning Simulations)

CAMELS

Villaescusa-Navarro+22

Ravanbakhsh+17, Brehmer+19, Ribli+19, Pan+19, Ntampaka+19, Alexander+20, Arjona+20, Coogan+20, Escamilla-Rivera+20, Hortua+20, Vama+20, Vernardos+20, Wang+20, Mao+20, Arico+20, Villaescusa_navarro+20, Singh+20, Park+21, Modi+21, Villaescusa-Navarro+21ab, Moriwaki+21, DeRose+21, Makinen+21, Villaescusa-Navaroo+22

Training deeper networks

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING THE STEEPEST DESCENT

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING THE STEEPEST DESCENT

THE CHOICES OF THE LEARNING RATES AND THE FREQUENCY OF WEIGTH UPDATES ARE IMPORTANT HYPER PARAMETERS TO MAKE NNs MORE ROBUST

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING THE STEEPEST DESCENT

THE CHOICES OF THE INITIAL WEIGHTS AND THE LEARNING RATES ARE IMPORTANT

LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$

THERE ARE DIFFERENT WAYS
TO UPDATE THE LEARNING RATE

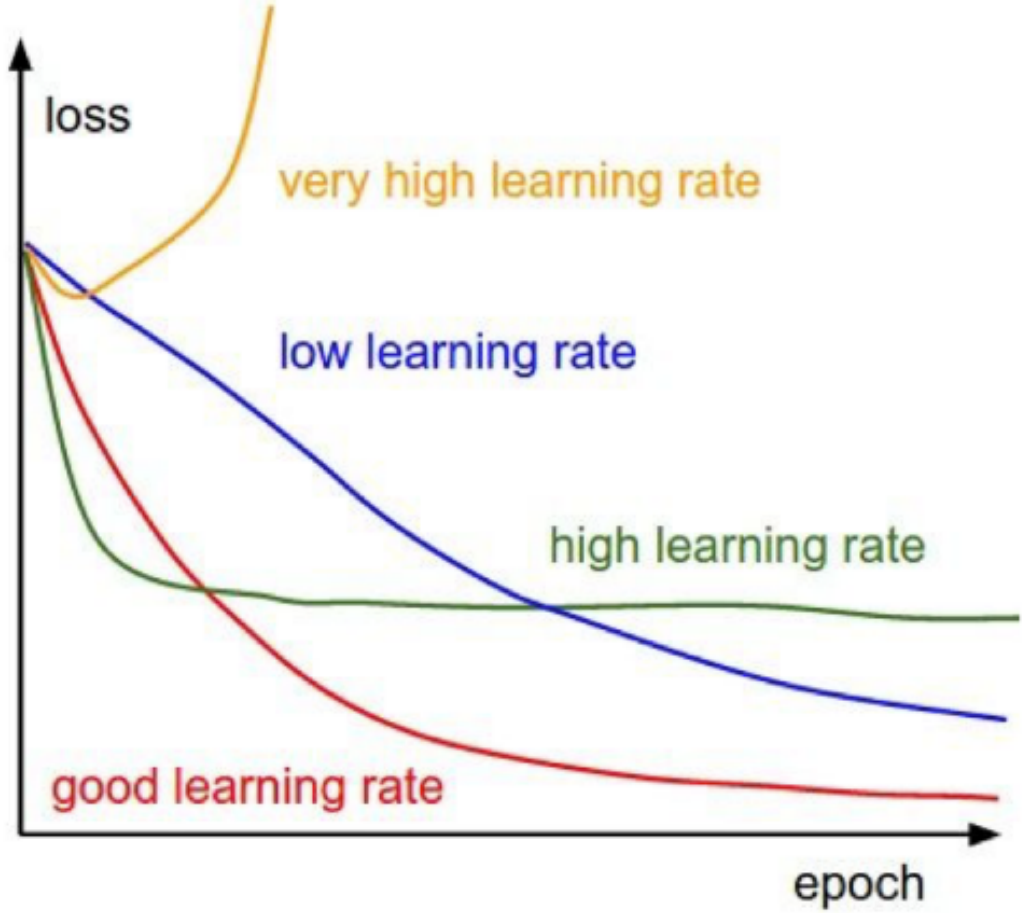
Credit:

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING THE STEEPEST DESCENT

THE CHOICES OF THE INITIAL WEIGHTS AND THE LEARNING RATES ARE IMPORTANT

WE WILL TALK ABOUT THIS LATER

LEARNING RATES



Credit:

LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$

THERE ARE DIFFERENT WAYS
TO UPDATE THE LEARNING RATE

ADAGRAD:

THE LEARNING RATE IS SCALED DEPENDING ON THE HISTORY OF PREVIOUS GRADIENTS

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{G_t + \epsilon}} \nabla f(W_t)$$

G IS A MATRIX CONTAINING ALL PREVIOUS GRADIENTS. WHEN THE GRADIENT BECOMES LARGE THE LEARNING RATE IS DECREASED AND VICE VERSA.

$$G_{t+1} = G_t + (\nabla f)^2$$

LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$

THERE ARE DIFFERENT WAYS
TO UPDATE THE LEARNING RATE

RMSPROP:

THE LEARNING RATE IS SCALED DEPENDING ON THE HISTORY OF PREVIOUS GRADIENTS

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{G_t + \epsilon}} \nabla f(W_t)$$

SAME AS **ADAGRAD** BUT G IS CALCULATED BY EXPONENTIALLY DECAYING AVERAGE

$$G_{t+1} = \lambda G_t + (1 - \lambda)(\nabla f)^2$$

ADAM [Adaptive moment estimator]:

SAME IDEA, USING FIRST AND SECOND ORDER MOMENTUMS

$$G_{t+1} = \beta_2 G_t + (1-\beta_2)(\nabla f)^2 \quad M_{t+1} = \beta_1 M_t + (1-\beta_1)(\nabla f)$$

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{\hat{G}_t + \epsilon}} \hat{M}_t$$

with: $\hat{M}_{t+1} = \frac{M_t}{1-\beta_1}$ $\hat{G}_{t+1} = \frac{G_t}{1-\beta_2}$

ADAM [Adaptive moment estimator]:

SAME IDEA, USING FIRST AND SECOND ORDER MOMENTUMS

$$G_{t+1} = \beta_2 G_t + (1-\beta_2)(\nabla f)^2 \qquad M_{t+1} = \beta_1 M_t + (1-\beta_1)(\nabla f)$$

ONLY FOR YOUR RECORDS

$$\sqrt{G_t + \epsilon}$$

with: $$\hat{M}_{t+1} = \frac{M_t}{1-\beta_1} \qquad \hat{G}_{t+1} = \frac{G_t}{1-\beta_2}$$

IN KERAS:

RMSprop [source]

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp optimizer.

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

Arguments

- **lr**: float >= 0. Learning rate.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

References

- rmsprop: Divide the gradient by a running average of its recent magnitude

IN KERAS:

Adam [source]

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam optimizer.

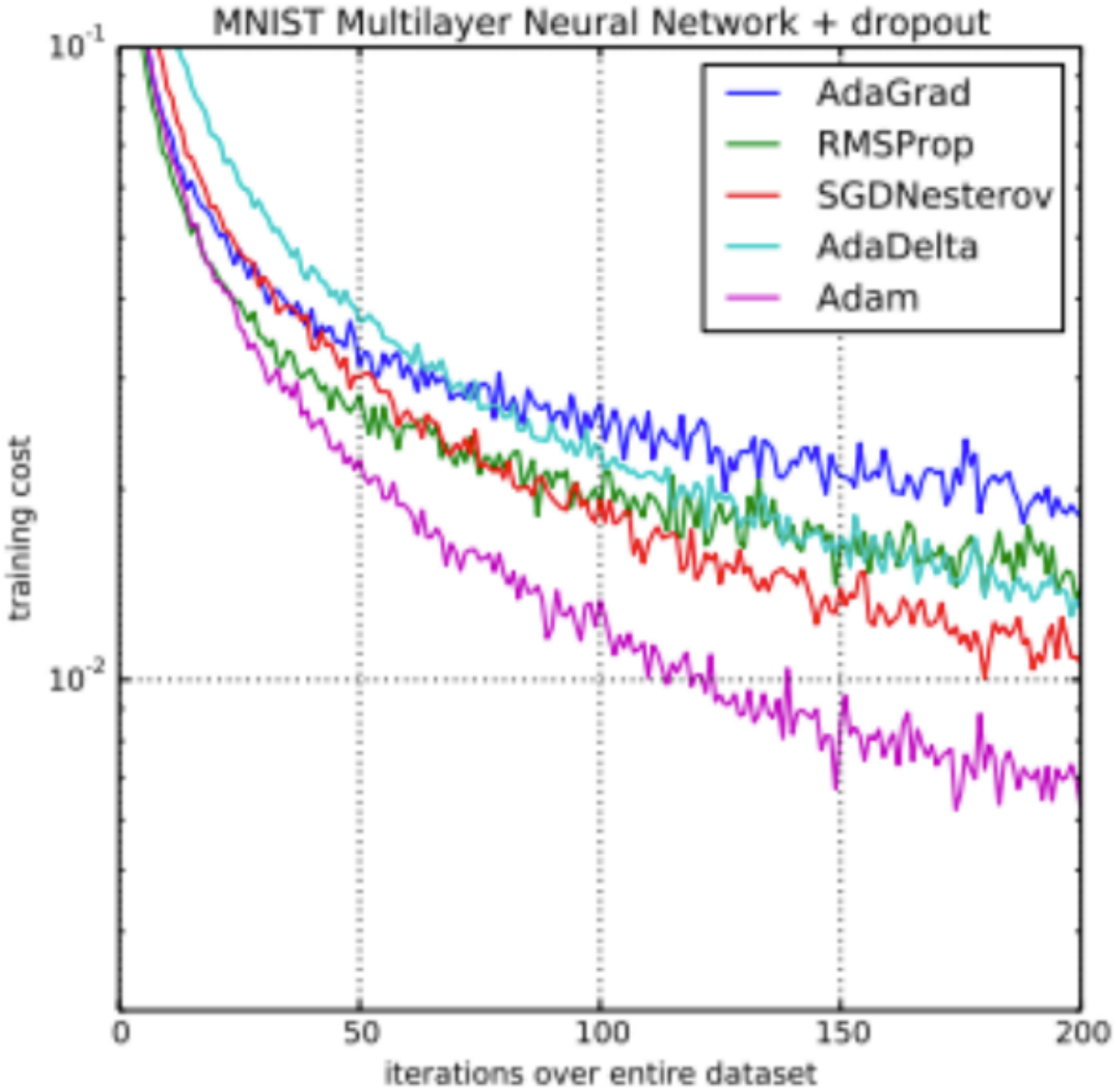
Default parameters follow those provided in the original paper.

Arguments

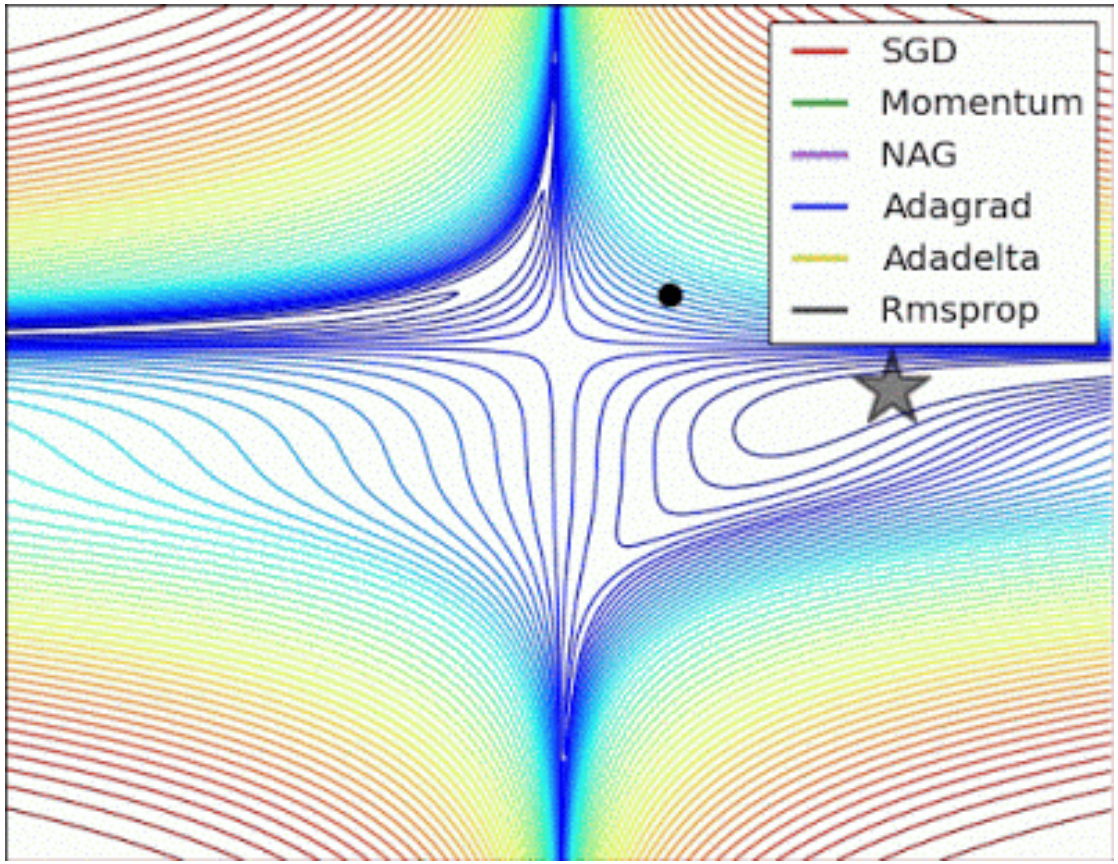
- **lr**: float >= 0. Learning rate.
- **beta_1**: float, 0 < beta < 1. Generally close to 1.
- **beta_2**: float, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.
- **amsgrad**: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

References

- Adam - A Method for Stochastic Optimization
- On the Convergence of Adam and Beyond


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset

Credit



Credit

BATCH GRADIENT DESCENT

LOCAL MINIMA CAN ALSO BE AVOIDED BY COMPUTING THE GRADIENT IN SMALL BATCHES INSTEAD OF OVER THE FULL DATASET

BATCH GRADIENT DESCENT

LOCAL MINIMA CAN ALSO BE AVOIDED BY COMPUTING THE GRADIENT IN SMALL BATCHES INSTEAD OF OVER THE FULL DATASET

MINI-BATCH GRADIENT DESCENT

$$V_{t+1/num} = W_t - \lambda_t \nabla f(W_t; x^{(i,i+b)}, y^{(i,i+b)})$$

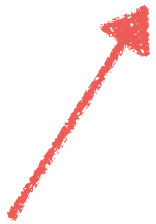
THE GRADIENT IS COMPUTED OVER A BATCH OF SIZE B

STOCHASTIC GRADIENT DESCENT

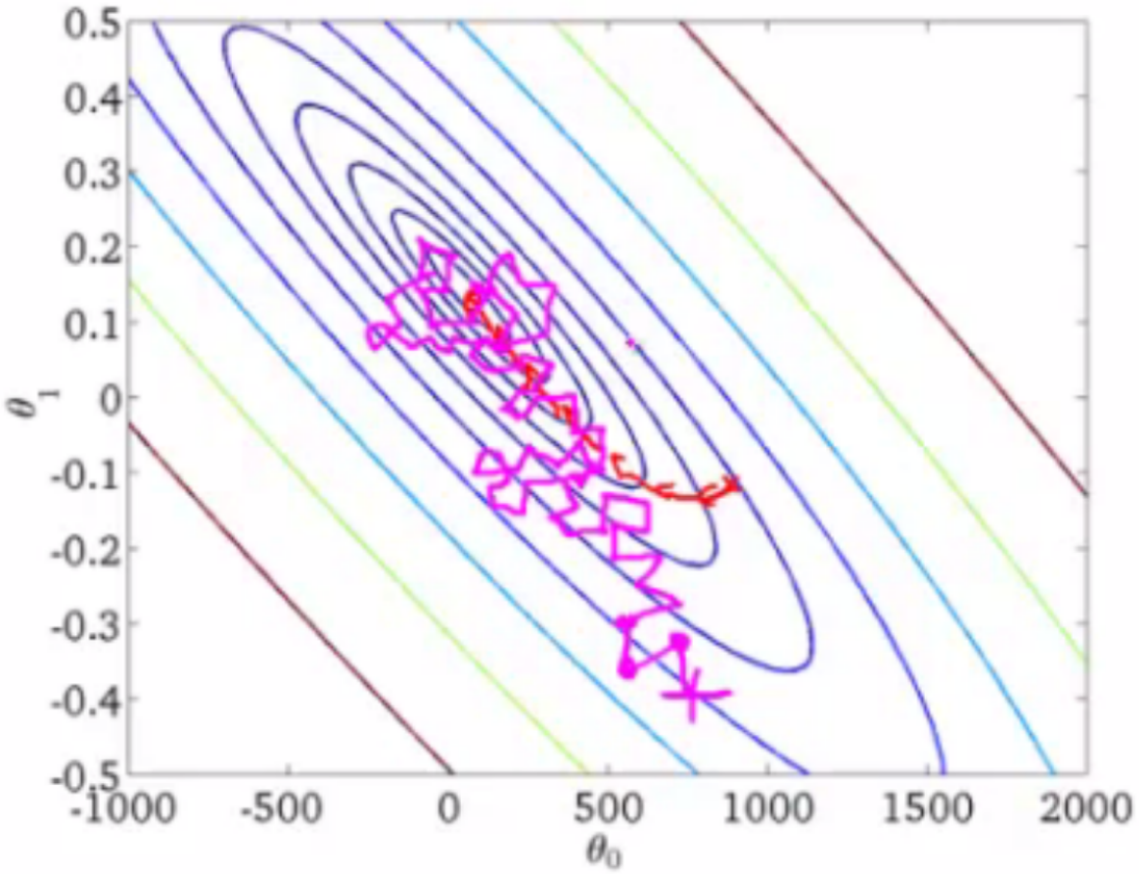
THE EXTREME CASE IS TO COMPUTE THE GRADIENT ON EVERY TRAINING EXAMPLE.

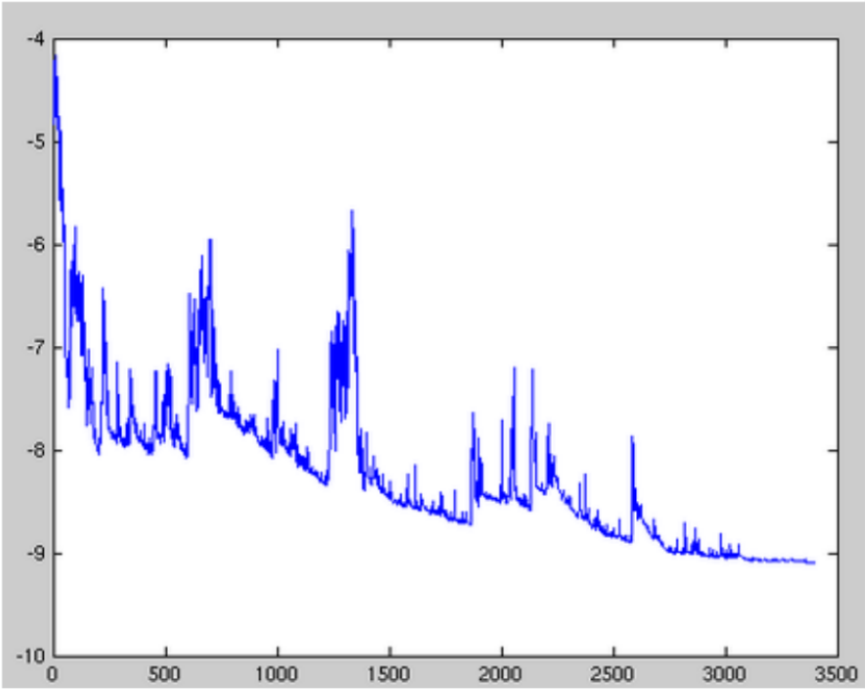
STOCHASTIC GRADIENT DESCENT

$$V_{t+1/num} = W_t - \lambda_t \nabla f(W_t; x^{(i,i+b)}, y^{(i,i+b)})$$



b=1

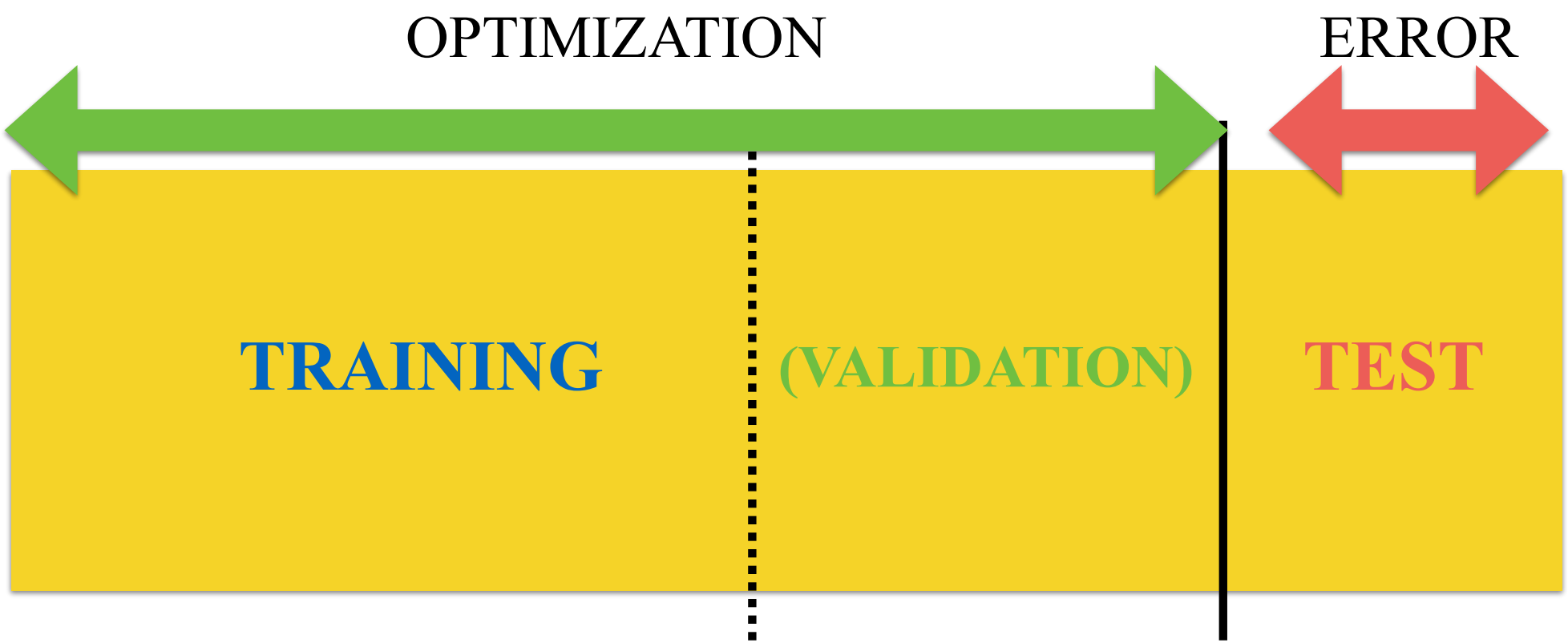




Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

Credit

IN PRACTICE



training set: use to train the classifier

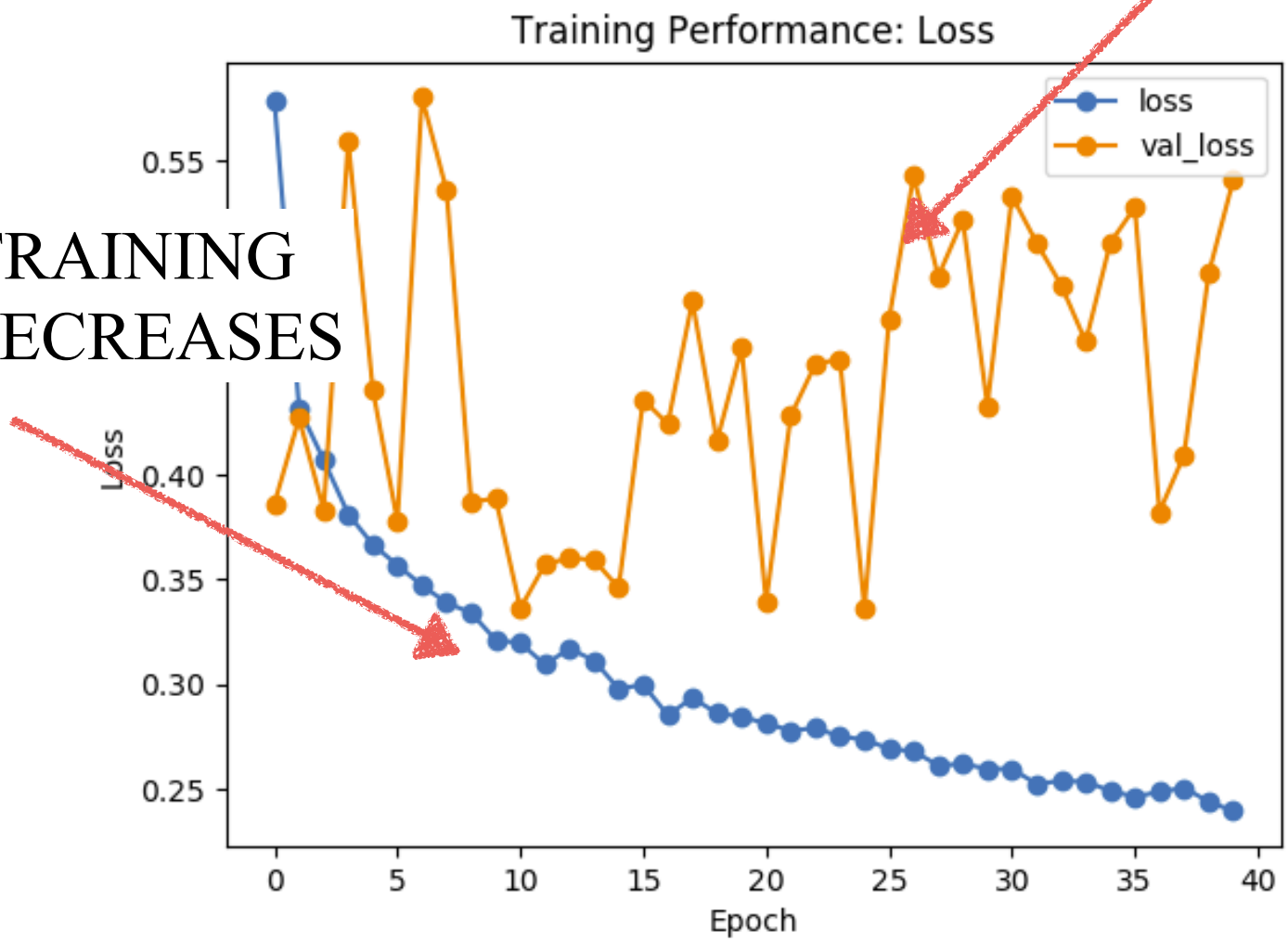
validation set: use to monitor performance in real time - check for overfitting

test set: use to train the classifier

OVER-FITTING

THE TEST STAYS CONSTANT OR INCREASES

THE TRAINING LOSS DECREASES

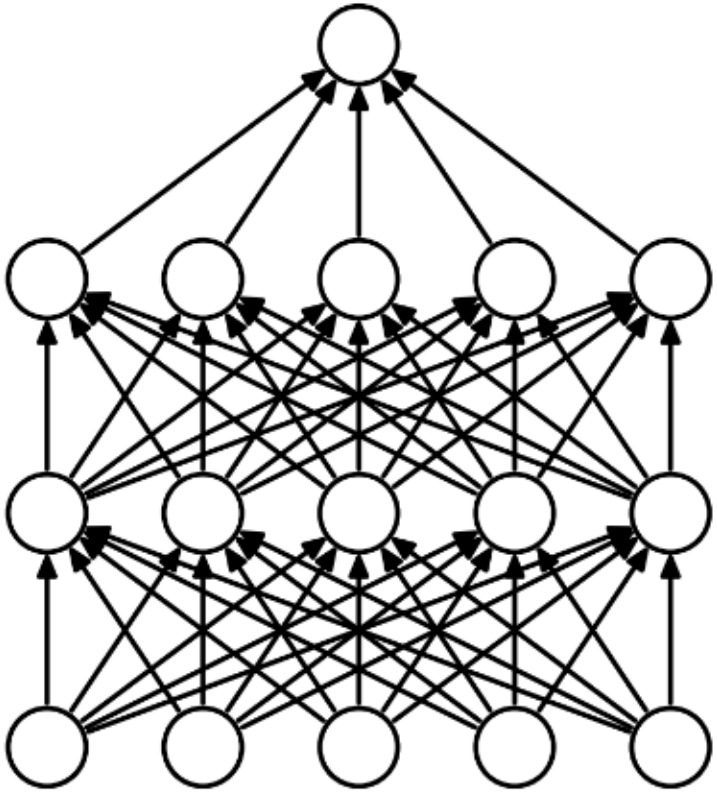


DROPOUT

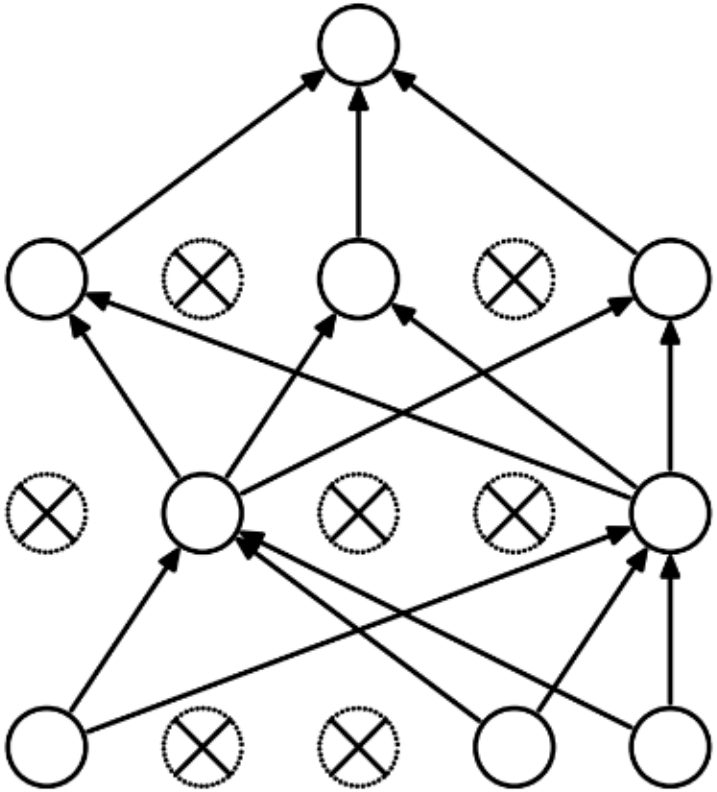
[Hinton+12]

- THE IDEA IS TO REMOVE NEURONS RANDOMLY DURING THE TRAINING

- ALL NEURONS ARE PUT BACK DURING THE TEST PHASE



(a) Standard Neural Net



(b) After applying dropout.

DROPOUT

WHY DOES IT WORK?

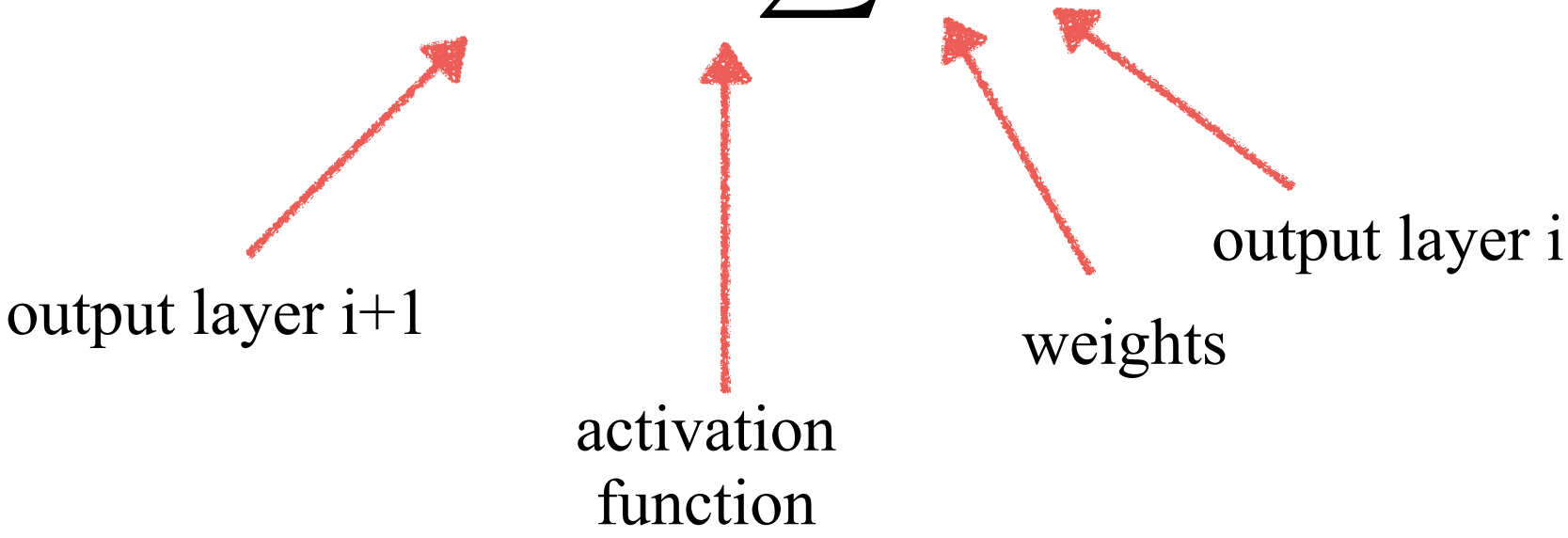
1. SINCE NEURONS ARE REMOVED RANDOMLY, IT AVOIDS CO-ADAPTATION AMONG THEMSELVES

2. DIFFERENT SETS OF NEURONS WHICH ARE SWITCHED OFF, REPRESENT A DIFFERENT ARCHITECTURE AND ALL THESE DIFFERENT ARCHITECTURES ARE TRAINED IN PARALLEL. FOR N NEURONS ATTACHED TO DROPOUT, THE NUMBER OF SUBSET ARCHITECTURES FORMED IS 2^N. SO IT AMOUNTS TO PREDICTION BEING AVERAGED OVER THESE ENSEMBLES OF MODELS.

VANISHING / EXPLODING GRADIENT PROBLEM

REMEMBER THAT:

$$y_{i+1} = \sigma[\sum w_i y_i]$$

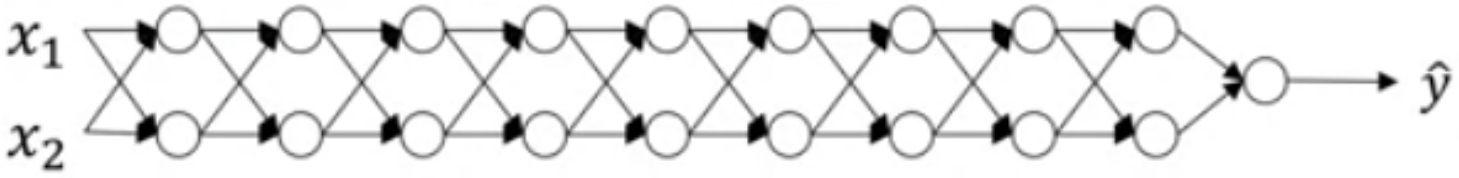


VANISHING / EXPLODING GRADIENT PROBLEM

WITH MANY LAYERS:

$$y_n = \sigma\left(...\sigma\left(...\sigma\left(\sum w_0 x\right)\right)\right)$$

VANISHING/EXPLODING GRADIENT PROBLEM



$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \qquad \hat{y} = x \prod_n w_i$$

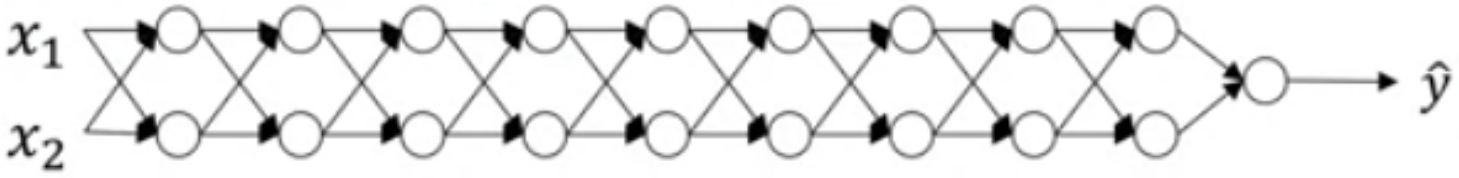
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

IF WEIGHTS ARE ALL INITIALIZED TO VALUES <<1:

$$\hat{y} \to 0$$

VANISHING GRADIENT

VANISHING/EXPLODING GRADIENT PROBLEM



$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \qquad \hat{y} = x \prod_n w_i$$

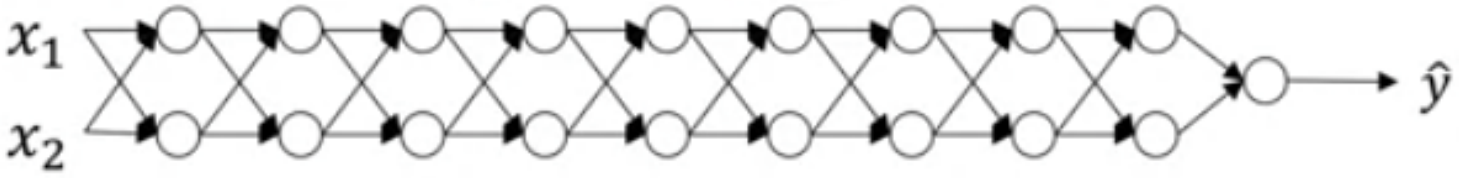
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

IF WEIGHTS ARE ALL INITIALIZED TO VALUES >1:

$$\hat{y} \to \infty$$

EXPLODING GRADIENT

VANISHING/EXPLODING GRADIENT PROBLEM



$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \qquad \hat{y} = x \prod_n w_i$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

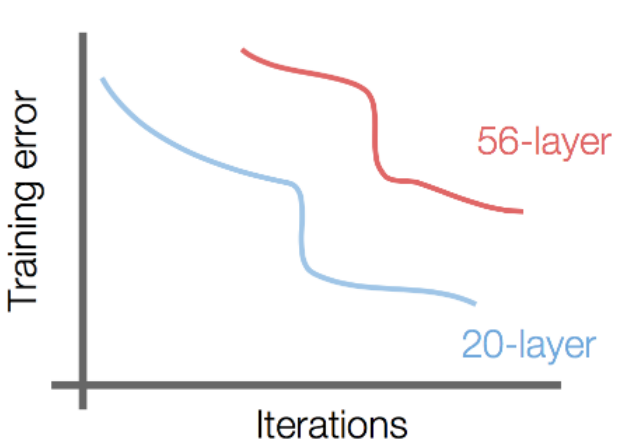
IF WEIGHTS ARE ALL INITIALIZED TO VALUES >1:

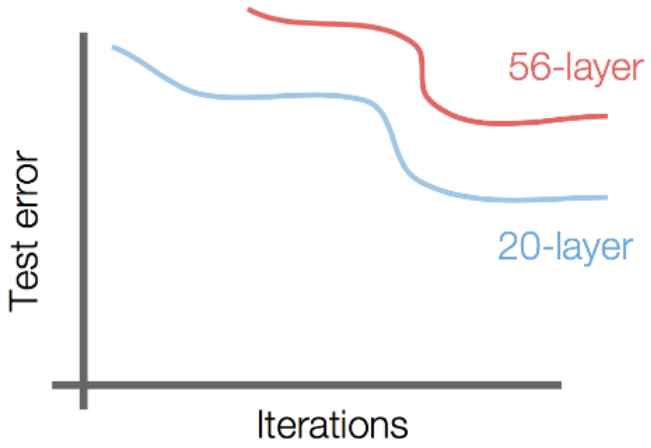
$$w_i^L \to \infty$$

EXPLODING GRADIENT

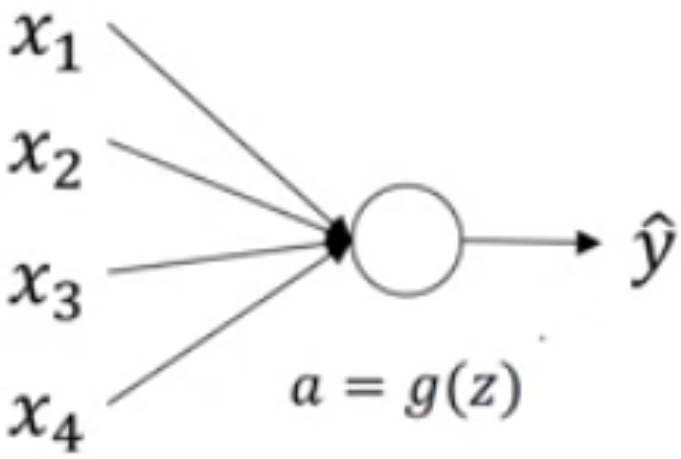
VANISHING/EXPLODING GRADIENT PROBLEM

TRAINING BECOMES UNSTABLE
VERY SLOW OR NO CONVERGENCE



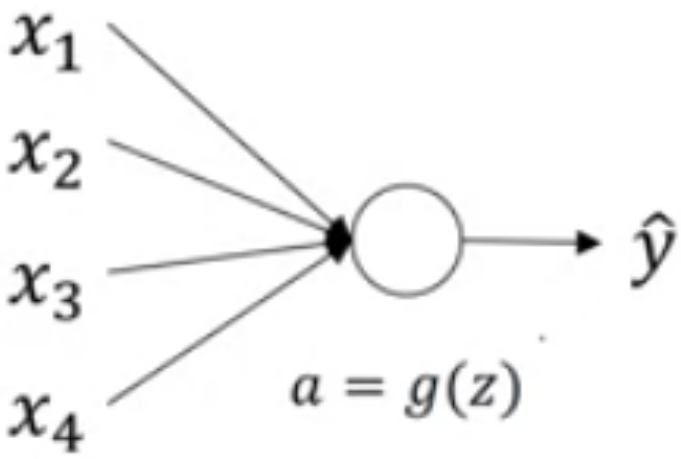


WEIGHT INITIALIZATION IS A KEY POINT…



$$z = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n$$

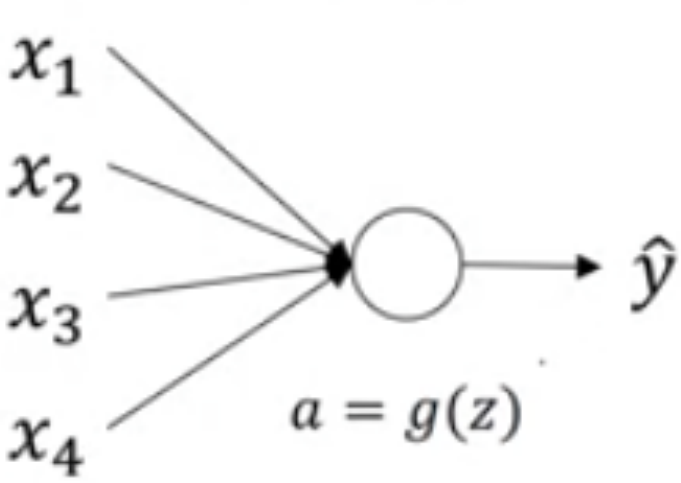
WEIGHT INITIALIZATION IS A KEY POINT…



$$z = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n$$

THE LARGER n, THE SMALLER WEIGHTS SHOULD BE…

WEIGHT INITIALIZATION IS A KEY POINT…



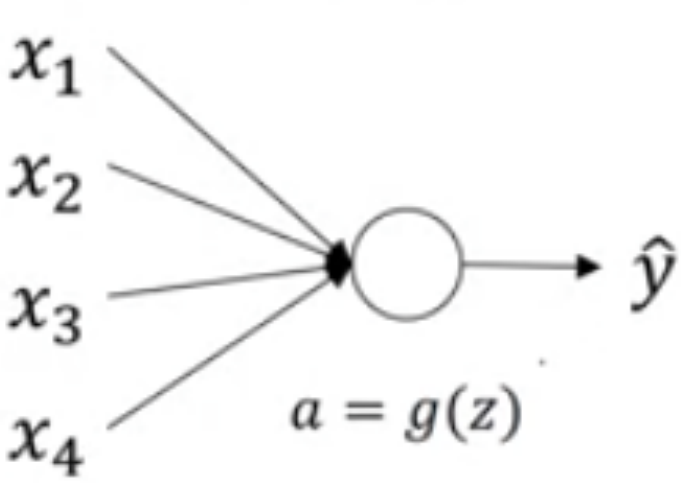
$$z = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n$$

THE LARGER n, THE SMALLER WEIGHTS SHOULD BE…

variance

ONE SIMPLE SOLUTION: $\sigma^2(w_i) = \frac{1}{n}$ ← number of inputs

WEIGHT INITIALIZATION IS A KEY POINT…



$$z = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n$$

THE LARGER n, THE SMALLER WEIGHTS SHOULD BE…

variance

ONE SIMPLE SOLUTION: $\sigma^2(w_i) = \frac{1}{n}$ ← number of inputs

WEIGHT INITIALIZATION IS A KEY POINT…

IMPLEMENTATION IN KERAS:

```
initialization = 'he_normal'
act = 'relu'

model = Sequential()
model.add(Convolution2D(depth, conv_size, conv_size, activation=act, border_mode='same',
name = "conv%i"%(layer_n), init=initialization, W_constraint=constraint))
```

WEIGHT INITIALIZATION IS A KEY POINT…

IMPLEMENTATION IN KERAS:

```
initialization = 'he_normal'
act = 'relu'

model = Sequential()
model.add(Convolution2D(depth, conv_size, conv_size, activation=act, border_mode='same',
name = "conv%i"%(layer_n), init=initialization, W_constraint=constraint))
```

MANY OTHER INITIALIZATIONS AVAILABLE:

`keras.initializers` → https://keras.io/initializers/

BATCH NORMALIZATION

[SZEGEDY+15]

A SOLUTION TO KEEP REASONABLE VALUES OF THE ACTIVATIONS IN DEEP NETWORKS

BATCH NORMALIZATION PREVENTS LOW OR LARGE VALUES BY RE-NORMALIZING THE VALUES BEFORE ACTIVATION FOR EVERY BATCH

INPUT

$$\hat{y}_i = \gamma \frac{y_i - E(y_i)}{\sigma(y_i)} + \beta$$

NORMALIZED INPUT

SCATTER

BATCH NORMALIZATION

[SZEGEDY+15]

BATCH NORMALIZATION SPEEDS UP AND STABILIZES TRAINING

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

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

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

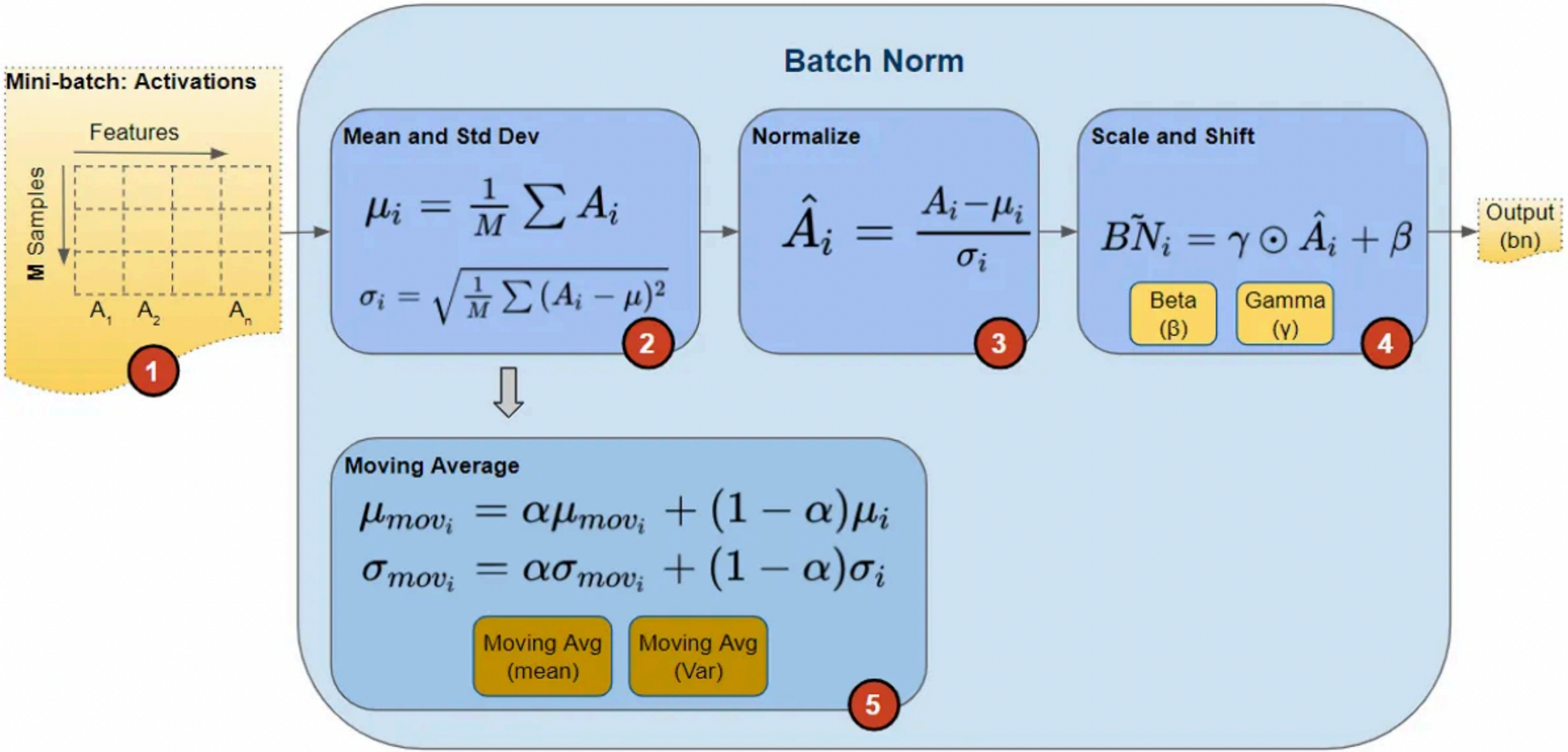
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

BATCH NORMALIZATION

[SZEGEDY+15]

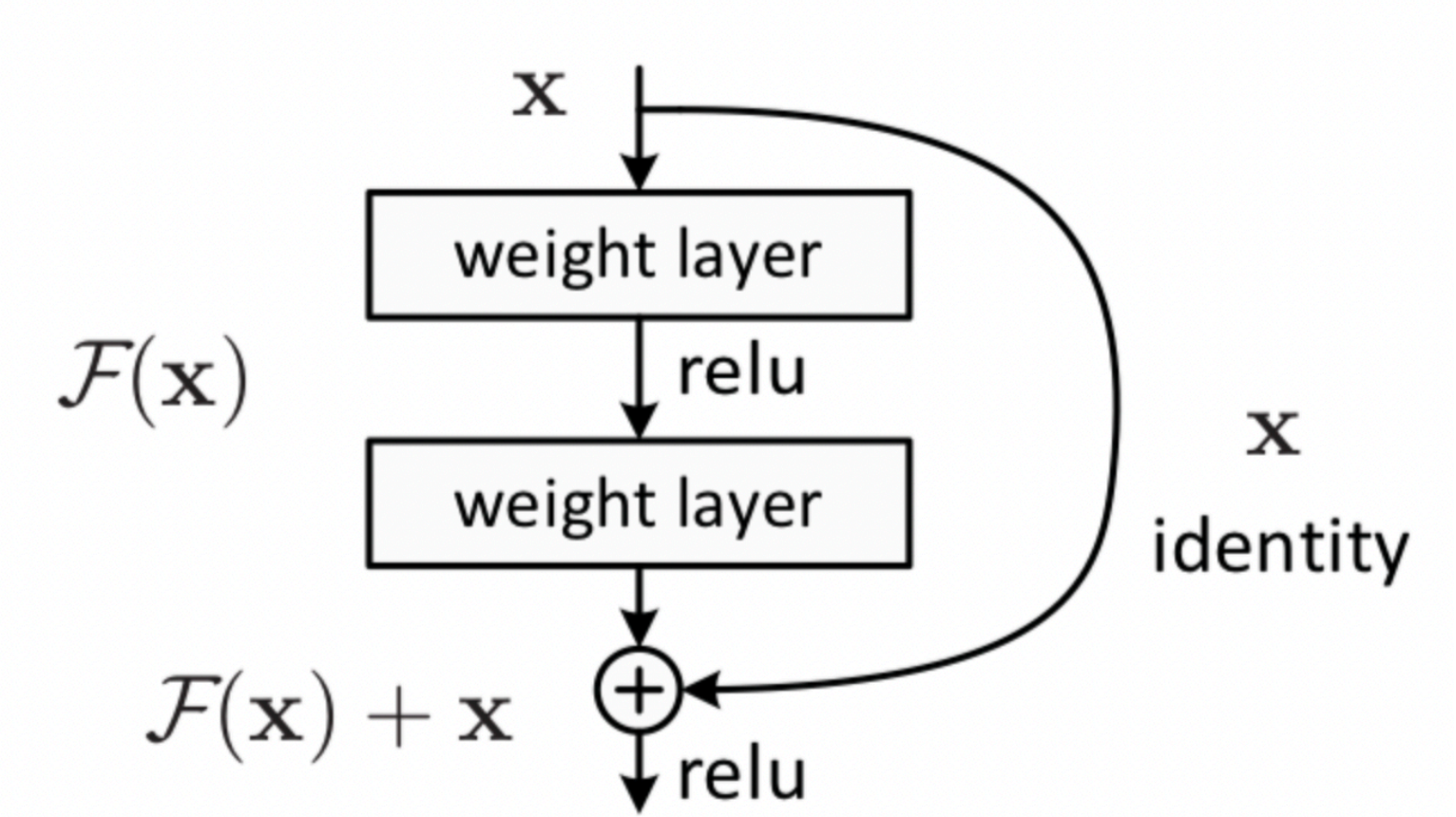


BATCH NORMALIZATION

[SZEGEDY+15]

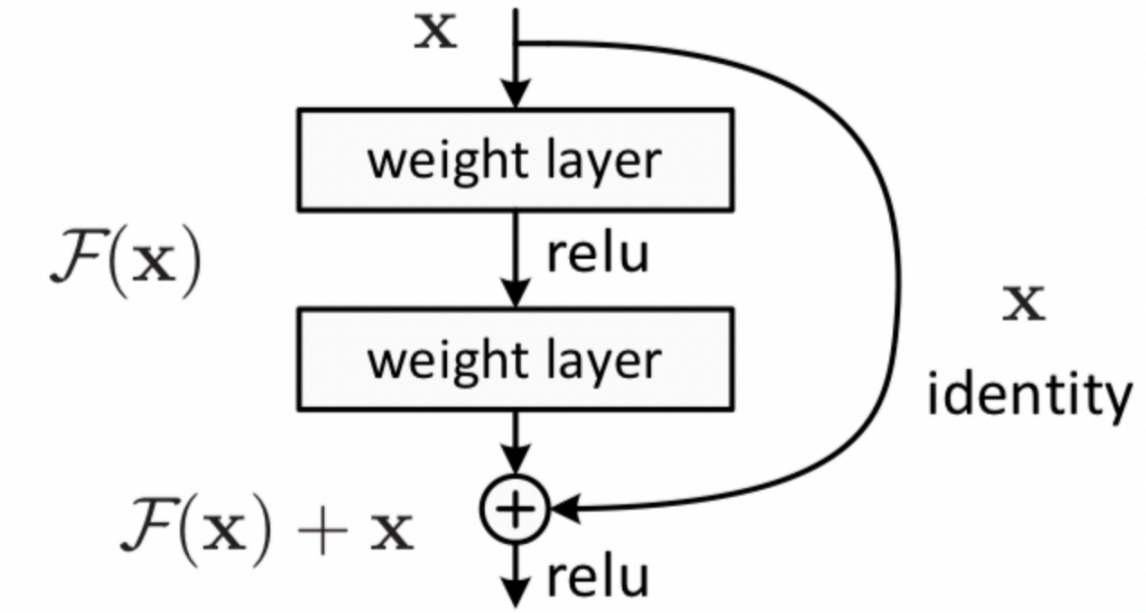
- Speeds up training
- Reduces internal covariate shift of the network
- Regularizes the network, prevents over fitting

RESIDUAL NETWORKS



Kaiming et al. 2015

RESIDUAL NETWORKS



- Adding additional / new layers would not hurt the model's performance as regularisation will skip over them if those layers were not useful.

- If the additional / new layers were useful, even with the presence of regularisation, the weights or kernels of the layers will be non-zero and model performance could increase slightly.

Kaiming et al. 2015