

HYPERPARAMETERS AND REGULARIZATION

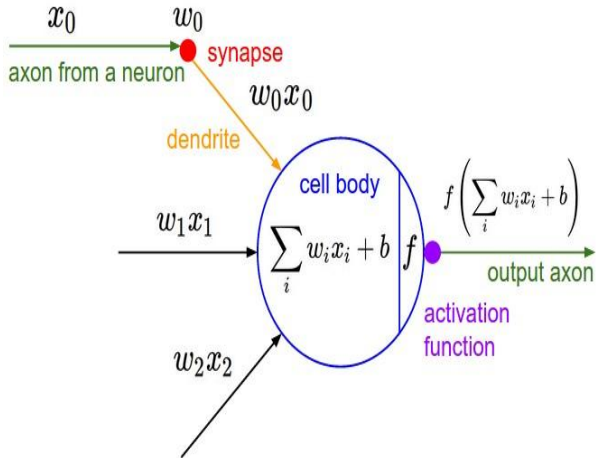


Analytical Index

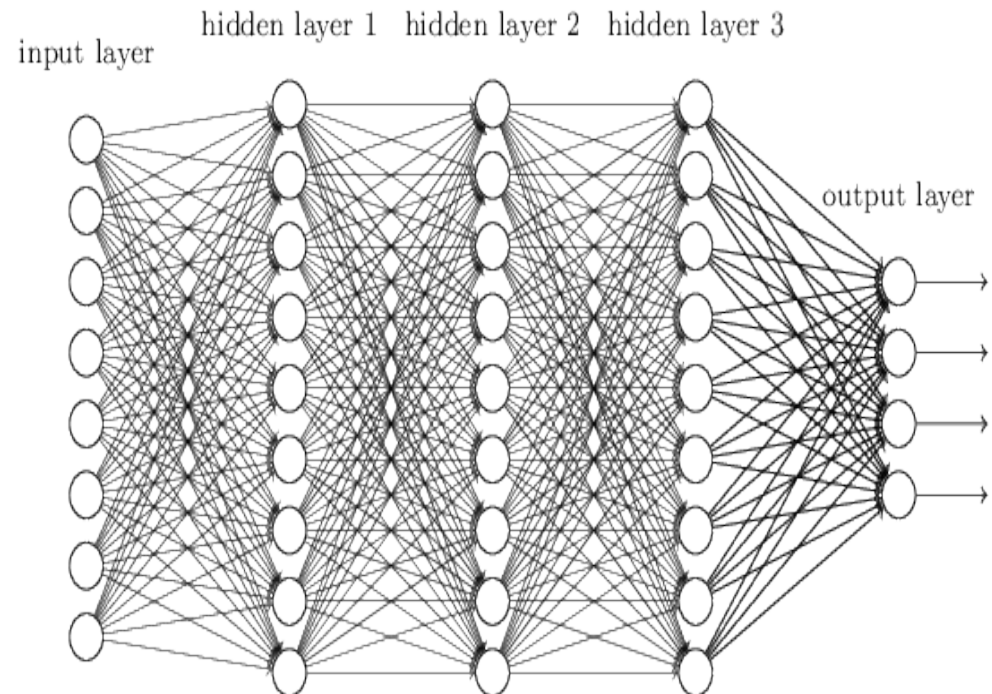
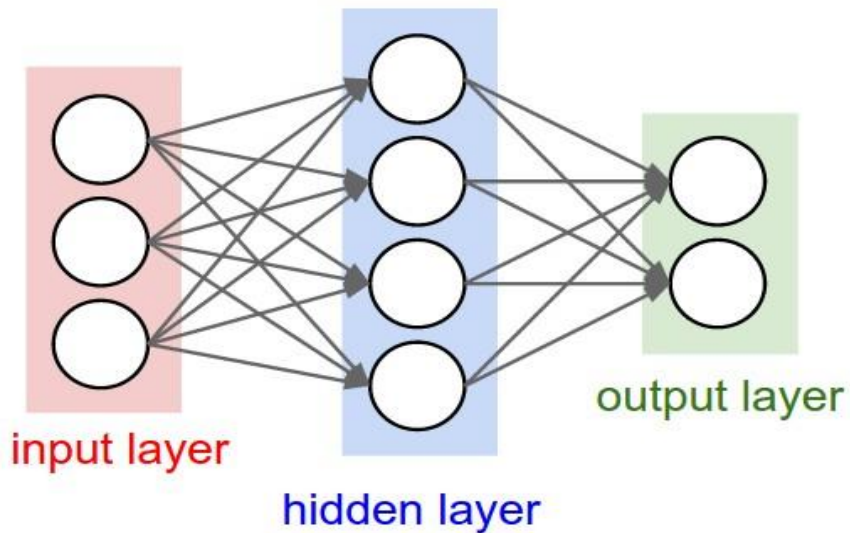
- 1. Neurons and Layers**
- 2. Activation**
- 3. Optimizers**
- 4. Cost Functions**
- 5. Regularization**
- 6. Data Augmentation**
- 7. Hyperparameter Optimization**

Neurons and Layers

Concepts review

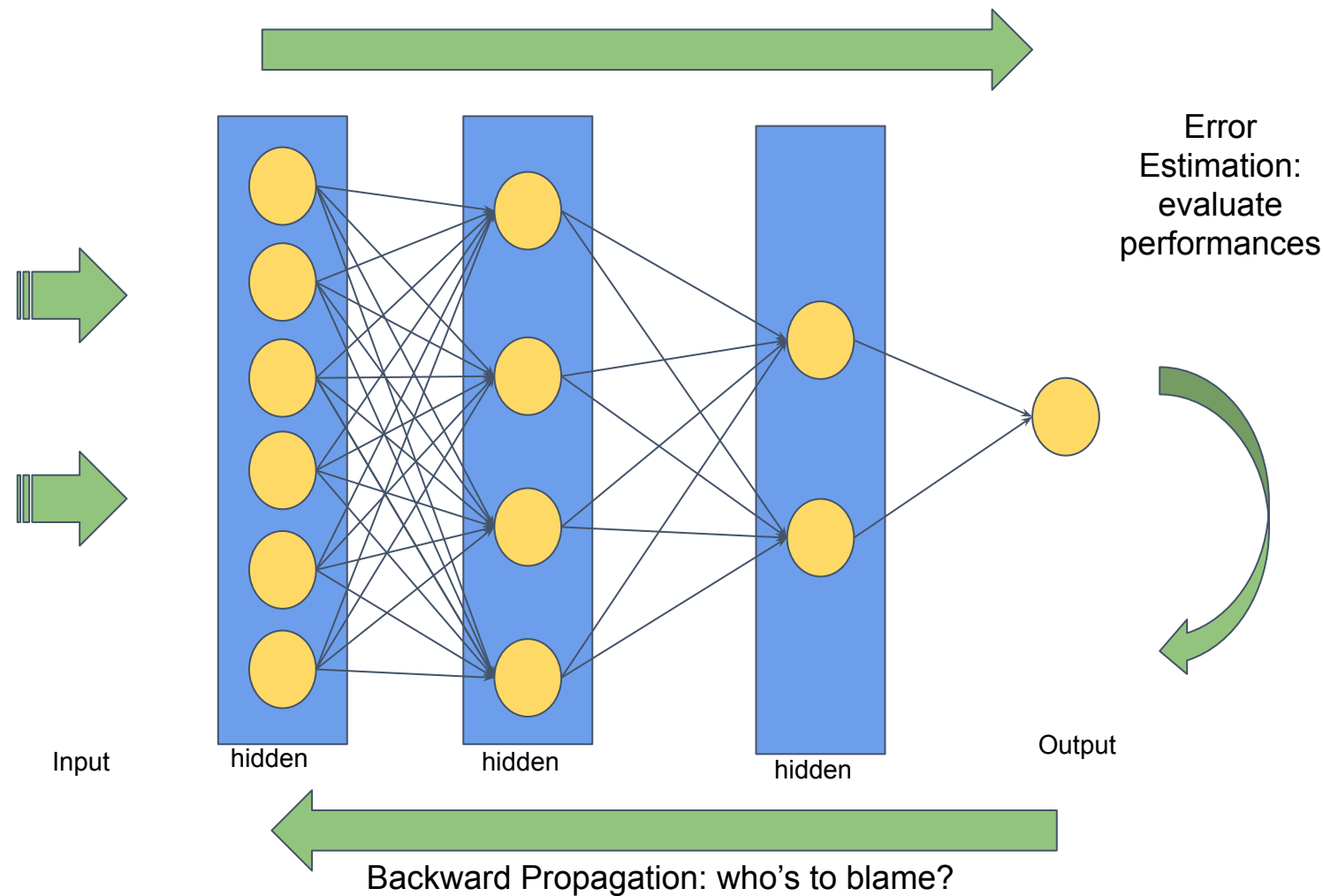


- What is a Layer. What about a Neuron?
- What do we know about weights?
- What is an activation function?



Backpropagation: high level picture

Forward Propagation: get a result

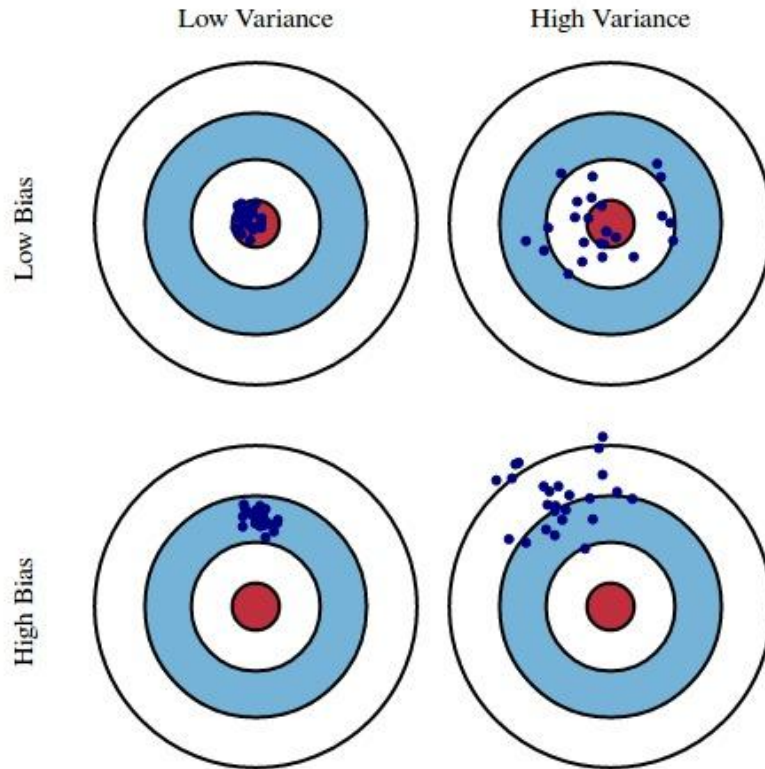


- A cost function C is defined
- Every parameter has its impact on the cost given some training examples
- Impacts are computed in terms of derivations

$$\frac{\partial C}{\partial w_{ij}}$$
- Use the chain rule to propagate error backwards

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_{ij}}$$

Overfitting

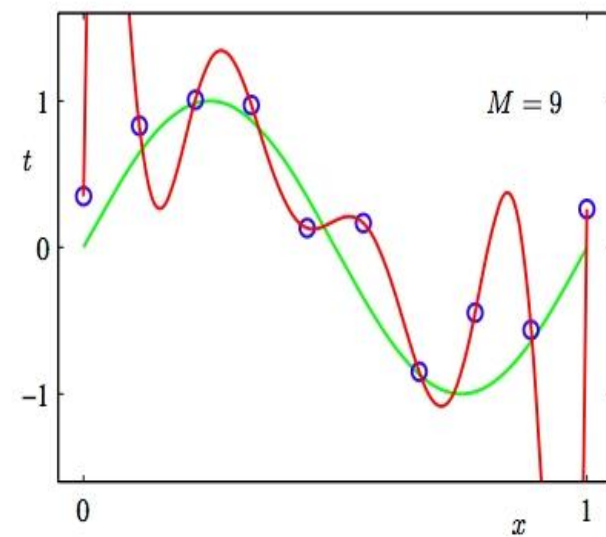
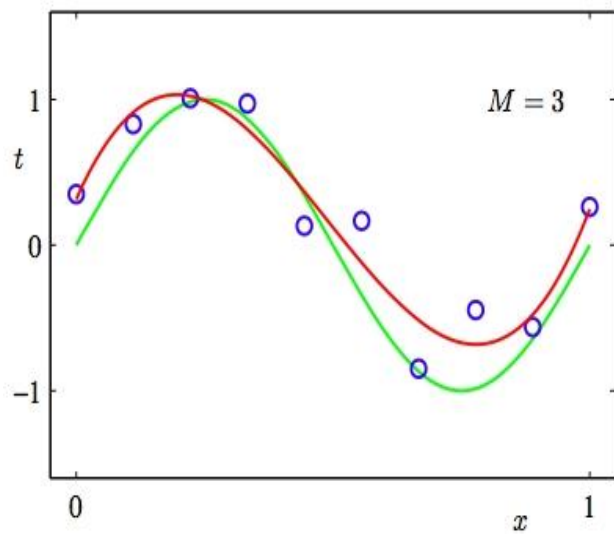
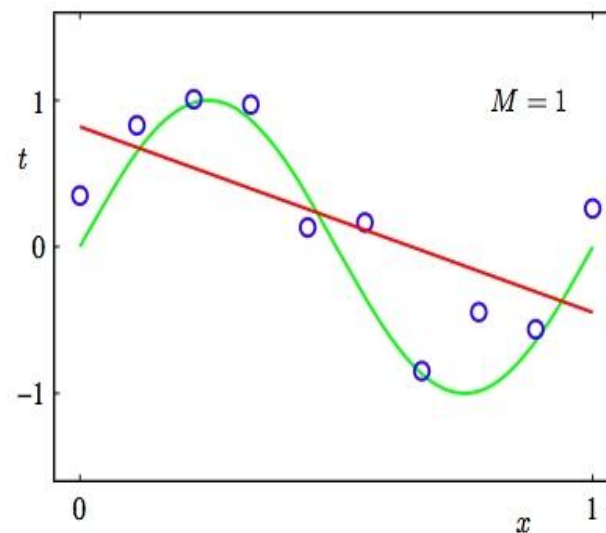
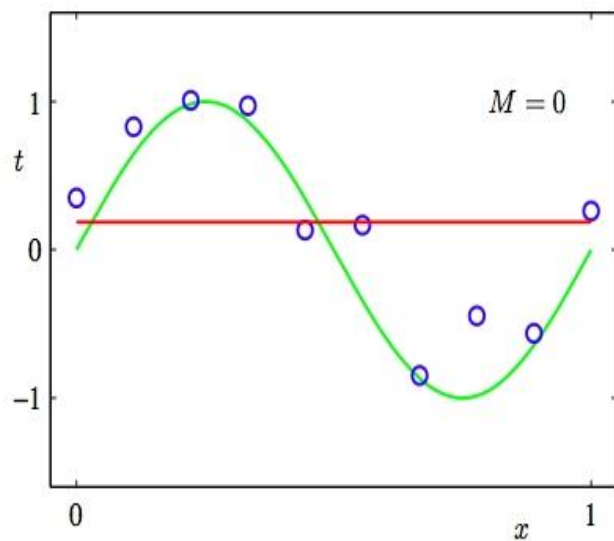


Trade off :

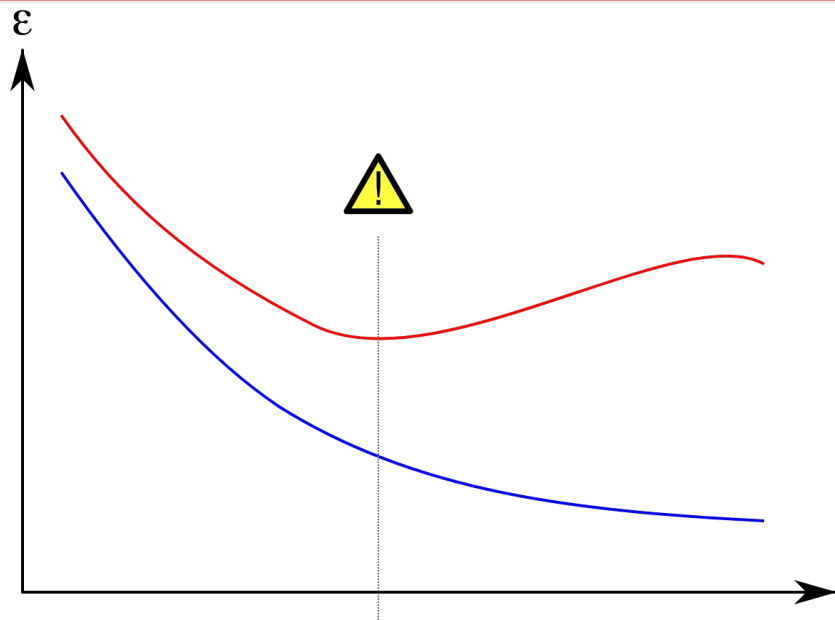
- **Overfitting :**
 - Low bias, High variance.
 - Complex, flexible model
- **Underfitting :**
 - High bias, Low Variance
 - Simple, rigid model.

An underfit model has high bias and low variance. Regardless of the specific samples in the training data, it cannot learn the problem. An overfit model has low bias and high variance. The model learns the training data too well and performance varies widely with new unseen examples or even statistical noise added to examples in the training dataset.

Overfitting



Overfitting and complexity



An overfit model is easily diagnosed by monitoring the performance of the model during training by evaluating it on both a training dataset and on a holdout validation dataset. Graphing line plots of the performance of the model during training, called learning curves, will show a familiar pattern.

For example, line plots of the loss (that we seek to minimize) of the model on train and validation datasets will show a line for the training dataset that drops and may plateau and a line for the validation dataset that drops at first, then at some point begins to rise again.

There are two ways to approach an overfit model:

1. Reduce overfitting by training the network on more examples.
2. Reduce overfitting by changing the complexity of the network.

A benefit of very deep neural networks is that their performance continues to improve as they are fed larger and larger datasets. A model with a near-infinite number of examples will eventually plateau in terms of what the capacity of the network is capable of learning. A model can overfit a training dataset because it has sufficient capacity to do so. Reducing the capacity of the model reduces the likelihood of the model overfitting the training dataset, to a point where it no longer overfits.

The capacity of a neural network model, its complexity, is defined by both its structure in terms of nodes and layers and the parameters in terms of its weights.

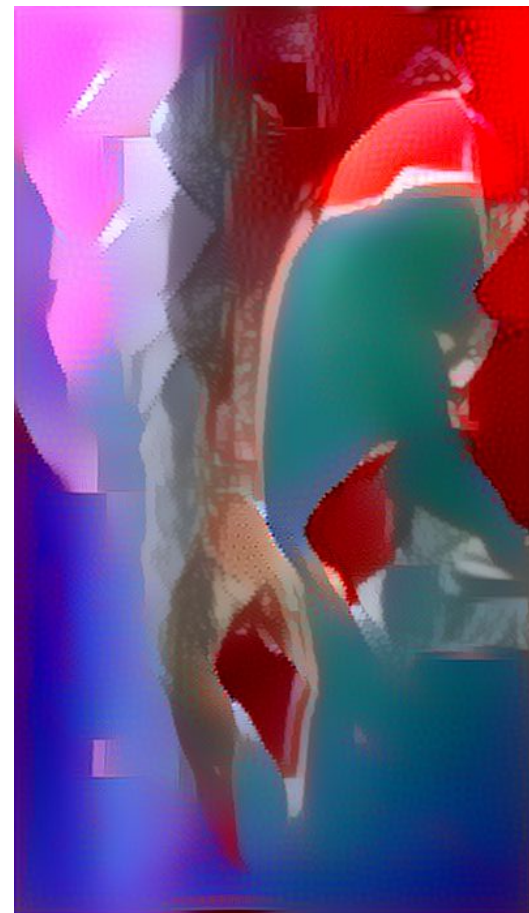
Hyperparameters

A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data.

- They are often used in processes to help estimate model parameters.
- They are often specified by the practitioner.
- They can often be set using heuristics.
- They are often tuned for a given predictive modeling problem.

We cannot know the best value for a model hyperparameter on a given problem. We may use rules of thumb, copy values used on other problems, or search for the best value by trial and error.

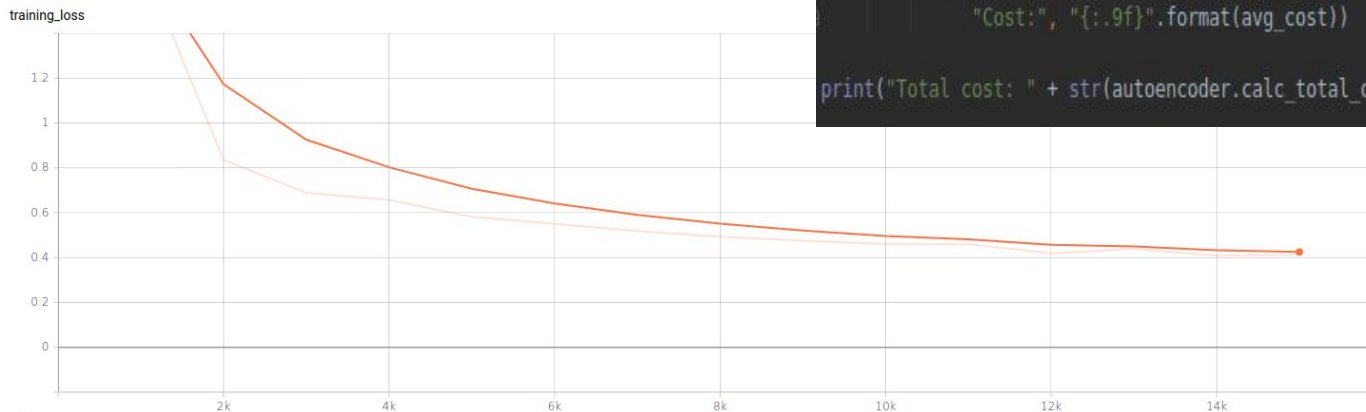
When a machine learning algorithm is tuned for a specific problem, such as when you are using a grid search or a random search, then you are tuning the hyperparameters of the model or order to discover the parameters of the model that result in the most skillful predictions.



Epochs

An epoch is one cycle through the full training dataset. Usually, training a neural network takes more than a few epochs. In other words, if we feed a neural network the training data for more than one epoch in different patterns, we hope for a better generalization when given a new "unseen" input (test data). An epoch is often mixed up with an iteration. Iterations is the number of batches or steps through partitioned packets of the training data, needed to complete one epoch.

In order to find the right number of epochs to train the model, the training/validation loss is a good hint of when the model starts overfitting.



```
autoencoder = Autoencoder(n_layers=[784, 200],
                          transfer_function=tf.nn.softplus,
                          optimizer=tf.train.AdamOptimizer(learning_rate=0.001))

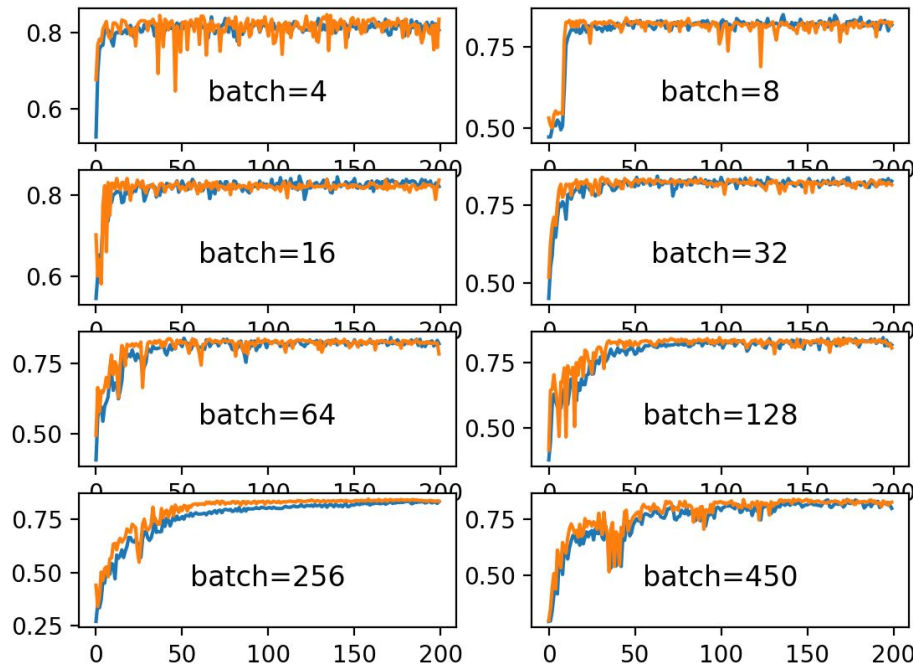
for epoch in range(training_epochs):
    avg_cost = 0.
    total_batch = int(n_samples / batch_size)
    # Loop over all batches
    for i in range(total_batch):
        batch_xs = get_random_block_from_data(X_train, batch_size)

        # Fit training using batch data
        cost = autoencoder.partial_fit(batch_xs)
        # Compute average loss
        avg_cost += cost / n_samples * batch_size

    # Display logs per epoch step
    if epoch % display_step == 0:
        print("Epoch:", '%d,' % (epoch + 1),
              "Cost:", "{:.9f}".format(avg_cost))

print("Total cost: " + str(autoencoder.calc_total_cost(X_test)))
```

Batch Size



The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.

We can think of a batch as a for-loop iterating over one or more samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.

A training dataset can be divided into one or more batches, but it is possible that a dataset does not divide evenly into the batch size. This can and does happen often when training a model. It simply means that the final batch has fewer samples than the other batches.

Parameter Initialization

Neural-net training essentially consists in repeating the two following steps:

- A forward step: “multiplying matrices”
- A backward step: update weights

During the forward step, the activations (and then the gradients) can quickly get really big or really small — this is due to the fact that we repeat a lot of matrix multiplications. Thus, to avoid exploding, weights should be in a good range. But what is a *good range* in practice? Quantitatively speaking, it implies having the output of the Matrix multiplications with the input vector produce an output vector (i.e. activations) with mean near 0 and standard deviation near 1. Then each layer will propagate these statistics across all the layers.

The Neural network needs to take into account those restrictions, but first it needs to have a proper initialization. In order to obtain that, there are some techniques like the Kalming initialization, the LSUV algorithm and the Xavier/Glorot initialization.

The Glorot initialization essentially draws samples from a truncated normal distribution centered on 0 with stddev = $\sqrt{2 / (\text{fan_in} + \text{fan_out})}$ where fan_in is the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor.

$$weights_{norm} \cdot \sqrt{\frac{2}{batch_size}}$$

Simplified ReLU version of Kalming Initialization

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

```

LSUV algorithm: Mishkin, Matas *All you need is a good initialization*

Activation

Activation Functions

Neuron preactivation:

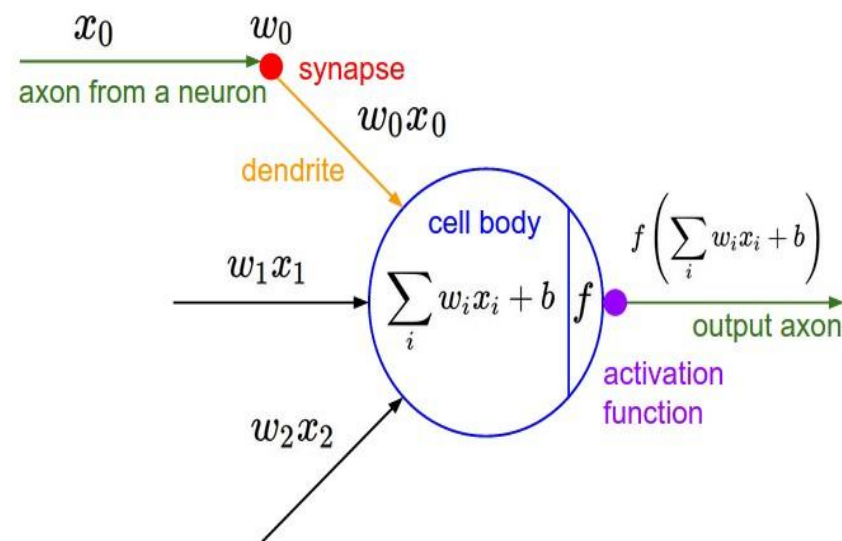
$$a(x) = b + \sum_i \omega_i \cdot x_i = b + \omega^t x$$

Neuron output activation:

$$h(x) = g(a(x)) = g\left(b + \sum_i \omega_i \cdot x_i\right)$$

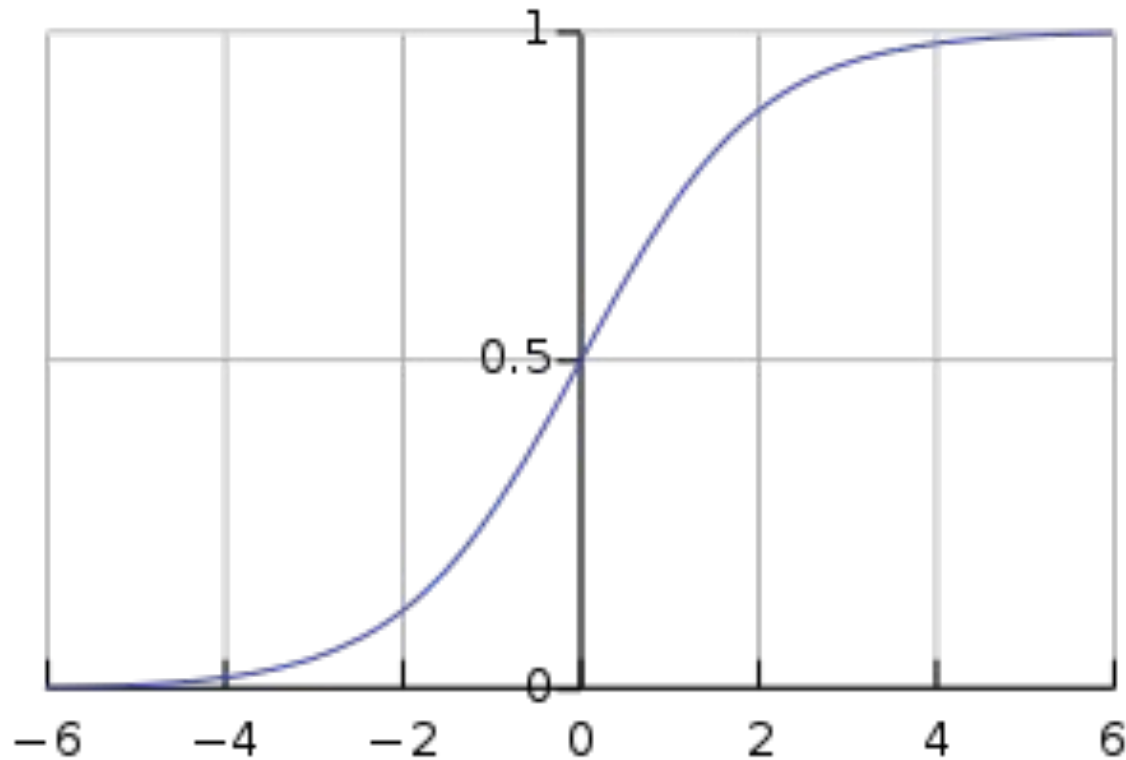
Where:

- ω are the weights (parameters)
- b is the bias
- $g(\cdot)$ is called **activation function**



Activation functions allow the network to learn non-linear behaviors in the data

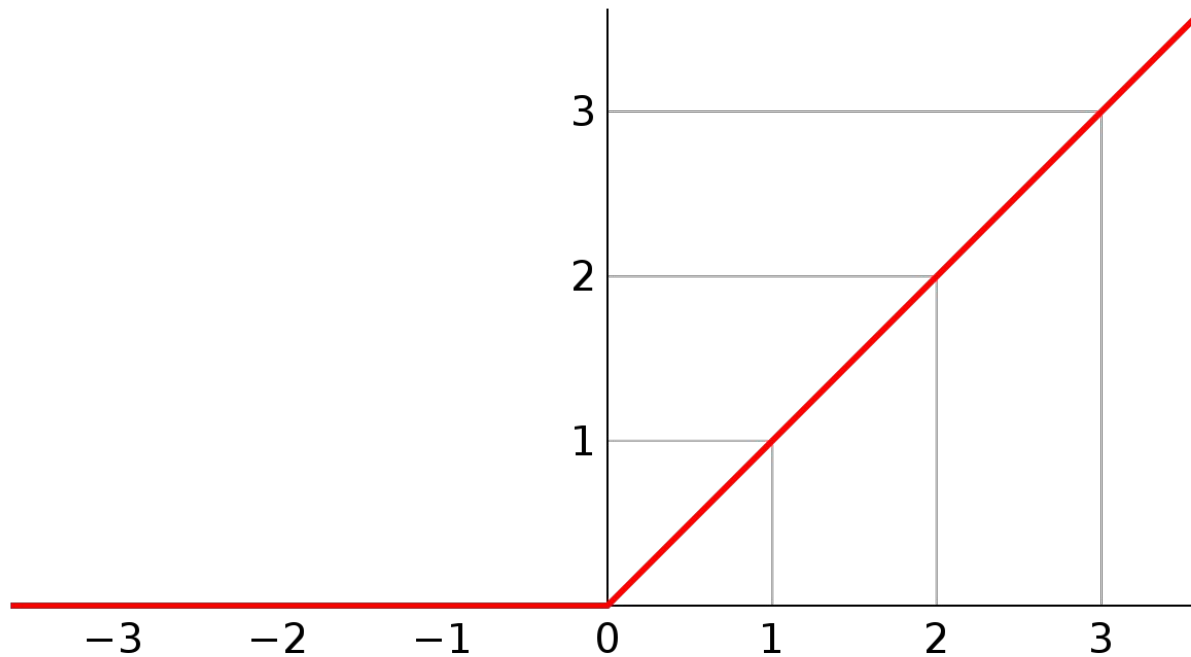
Sigmoid



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

- Bounded
- Probability-like function
- Dense computation
- Differentiable
- On many examples of fully connected layers

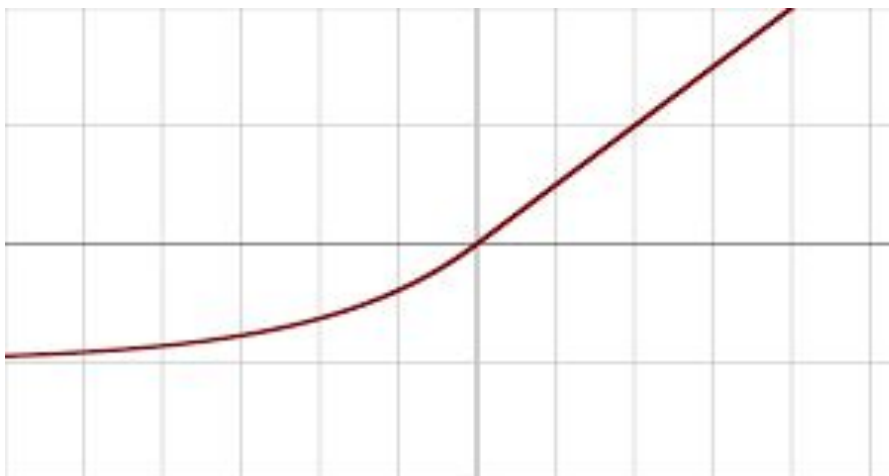
Rectified Linear Unit (Relu)



$$ReLU(x) = \max(0, x)$$

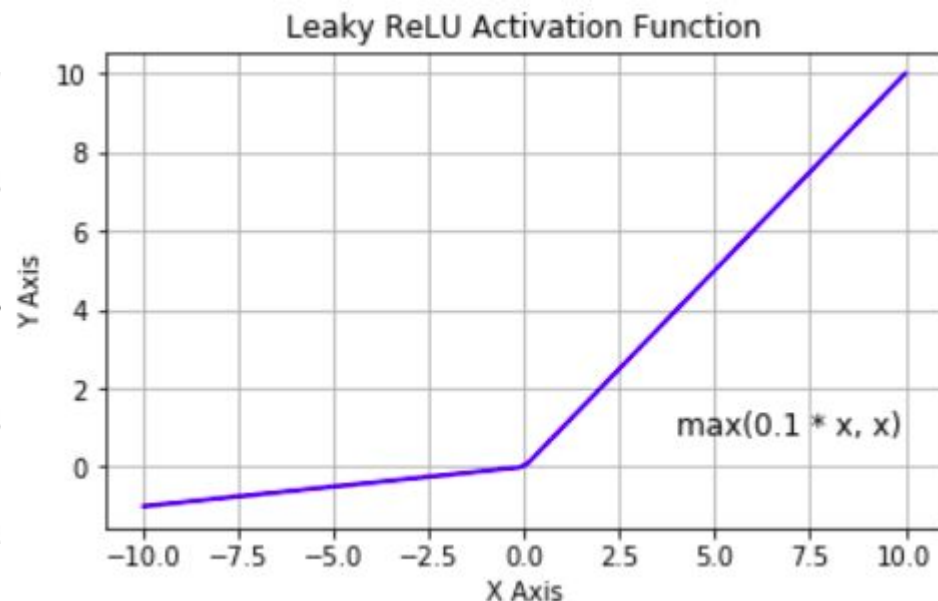
- Sparse activation
- Efficient computation
- “Differentiable”
- Unbounded
- Potential Dying Relu

ELU and Leaky ReLU



$$f(x) = \begin{cases} x & \text{if } x \geq 0, \\ a(e^x - 1) & \text{otherwise,} \end{cases}$$

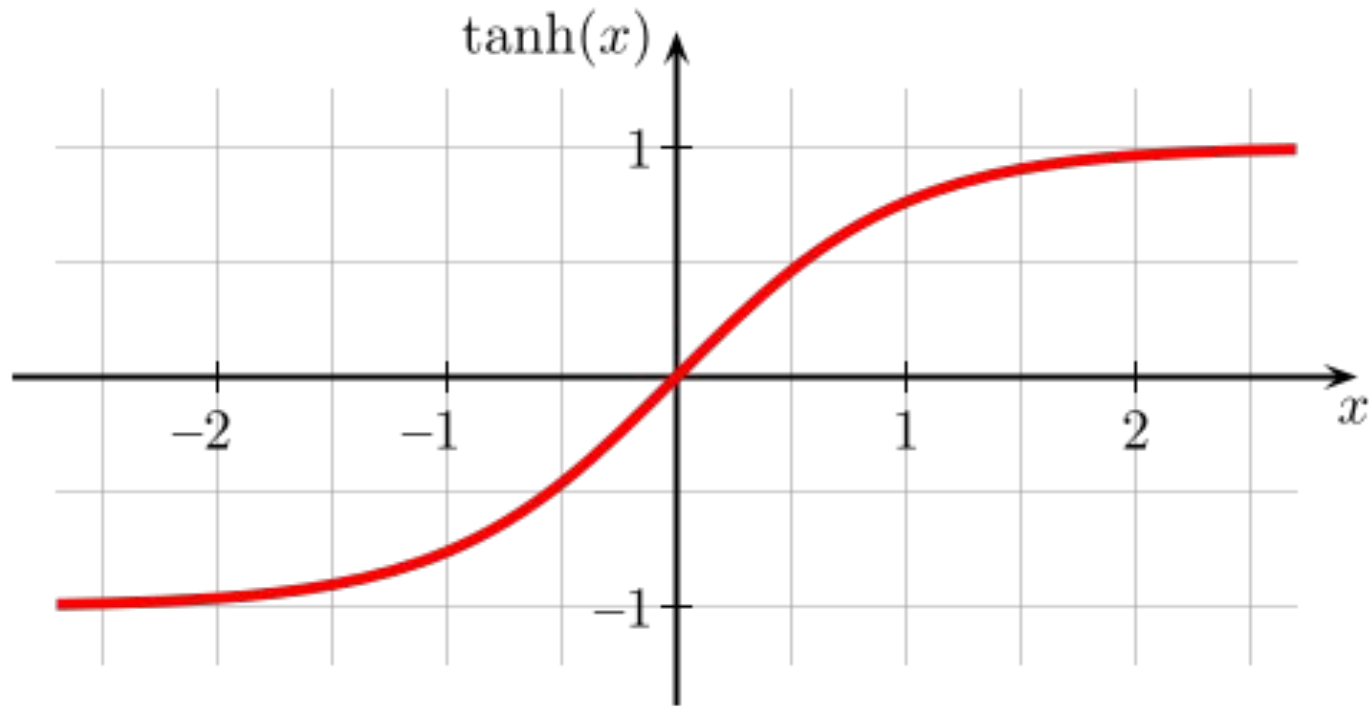
- ELUs are unbounded
- Tries to make the mean activations closer to zero, which speeds up learning.
- It has been shown that ELUs can obtain higher classification accuracy than ReLUs
- Differentiable
- Hyperparameter a needs to be positive or zero



$$f(x) = \begin{cases} x & \text{if } x \geq 0, \\ ax & \text{otherwise.} \end{cases}$$

- Leaky ReLUs are unbounded
- “Differentiable”
- Hyperparameter a is normally closer to zero: 0.01 or something similar

Hyperbolic tangent

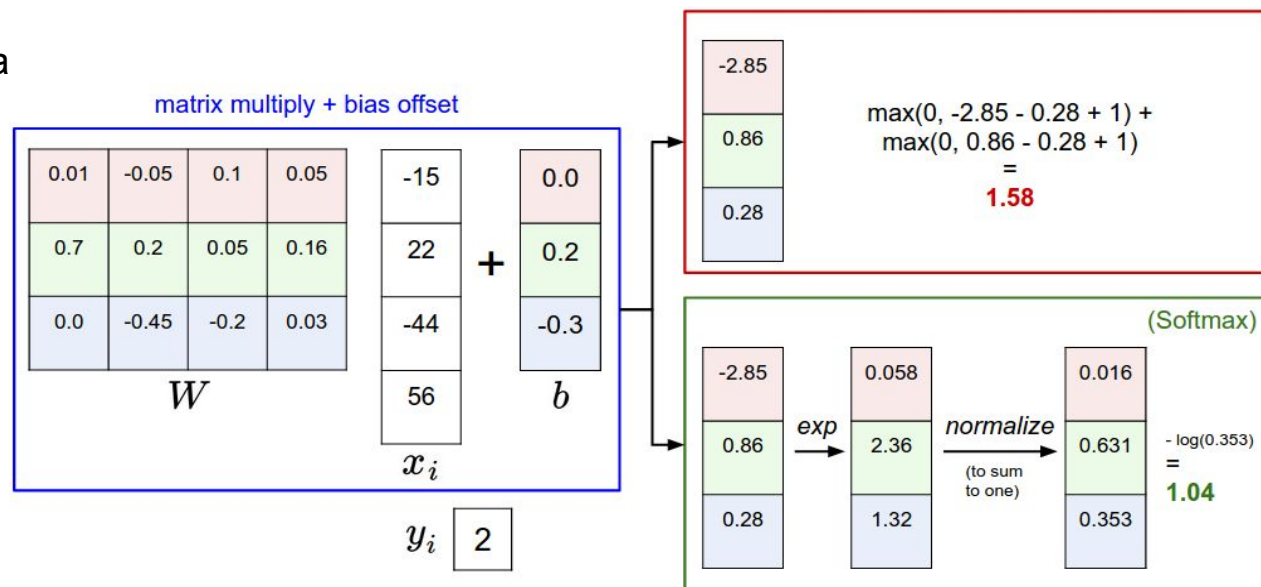


- Bounded
- Positive/negative values
- Dense computation
- Differentiable
- Bounded on $[-1, 1]$. Interesting when activation can be neutral, positive or negative (e.g: LSTM)

Softmax

- Represents probability on a categorical distribution
- Multiclass normalization
- Bounded
- Differentiable
- Used on final layers

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$



Some Easy Derivatives

- Sigmoid

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$$

- Hyperbolic Tangent

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

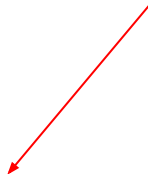
- Softmax

$$\frac{\partial \sigma(z)_i}{\partial z_j} = \begin{cases} \sigma(z)_j \cdot (1 - \sigma(z)_j) & \text{if } i = j, \\ \sigma(z)_i \cdot \sigma(z)_j, & \text{if } i \neq j \end{cases}$$

- ReLU

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x < 0, \\ \text{whaaaaaaaaaaaaaat????} & \text{if } x = 0 \end{cases}$$

Handset value



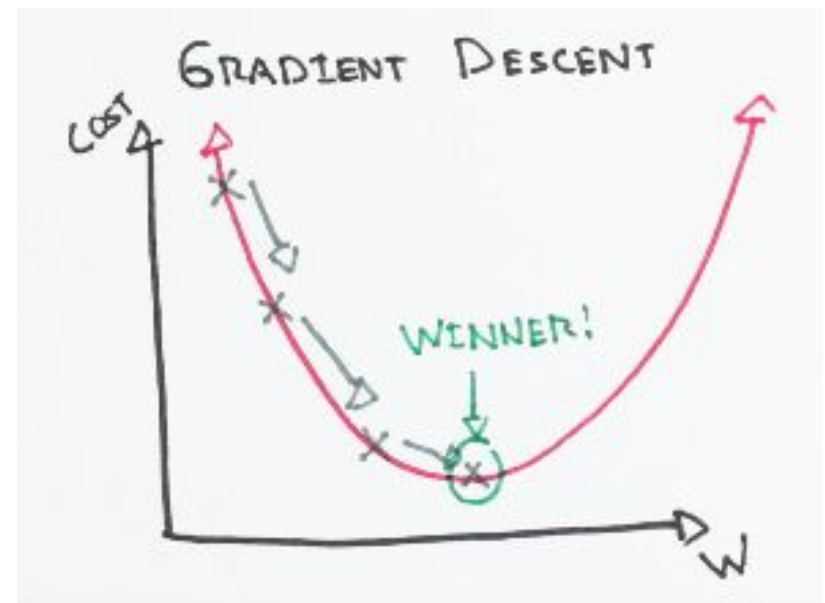
Optimizers

Optimizers

During the training process, the parameters (weights) of our model are tweaked and changed in order to minimize that loss function, and make the predictions as correct as possible. The precise way on what this is done is where optimizers come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold the model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Similarly, it's impossible to know what your model's weights should be right from the start. But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.



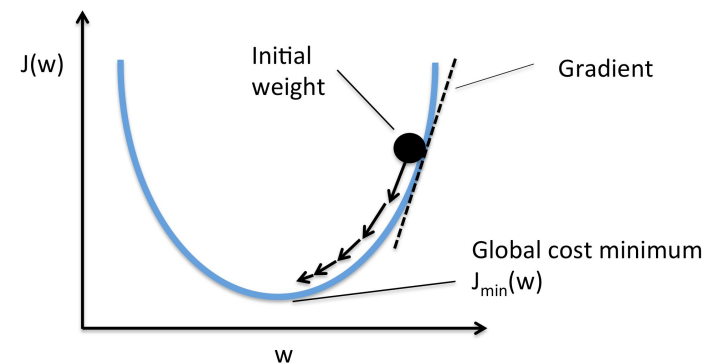
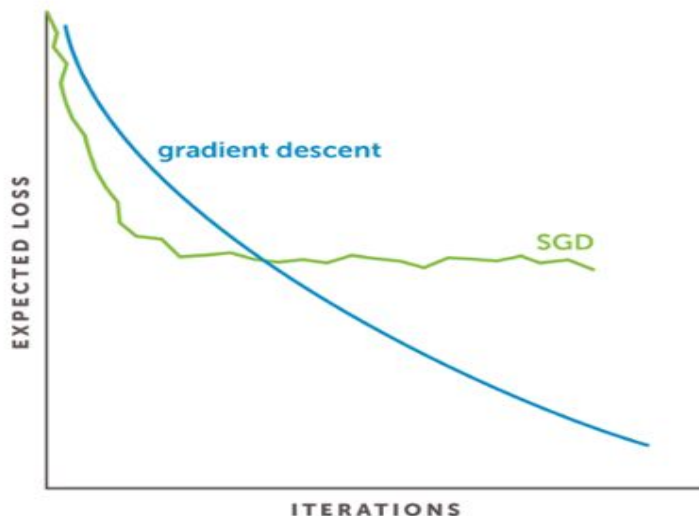
Stochastic Gradient Descent

Gradient Descent

- Goes over the whole training set.
- Very expensive.
- There isn't an easy way to incorporate new data to training set.

$$\hat{J}(\beta) = \frac{1}{m} \sum_i^m L(\hat{y}_i, y_i)$$

$$\beta^{i+1} \leftarrow \beta^i - \eta \nabla_{\beta} J(\beta^i)$$



Stochastic Gradient Descent (SGD)

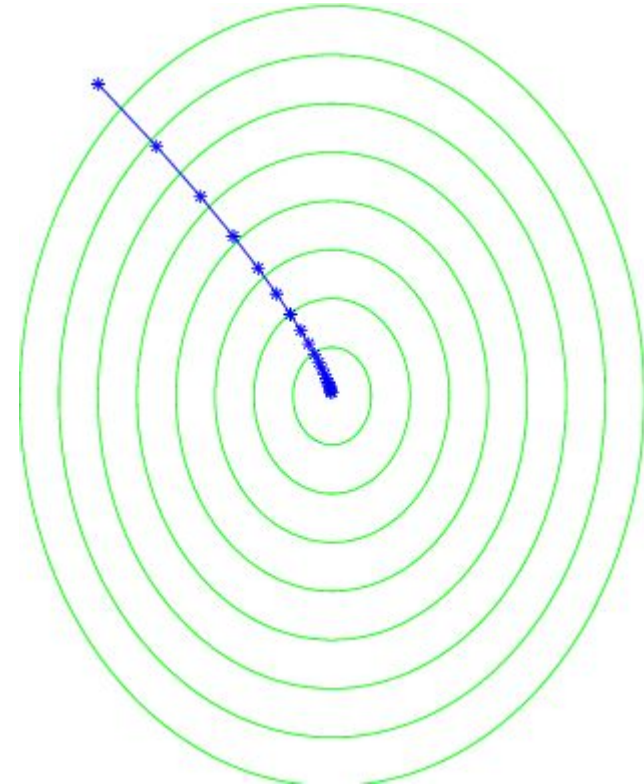
- Randomly sample a small number of examples (minibatch)
- Estimate cost function and gradient:

$$\hat{J}(\beta) = \frac{1}{k} \sum_i^k L(\hat{y}_i, y_i)$$

- **Batch size:** Length of the minibatch
- **Iteration:** Every time we update the weights
- **Epoch:** One pass over the whole training set.
- $k = 1 \Rightarrow$ online learning
- Small batches \Rightarrow regularization

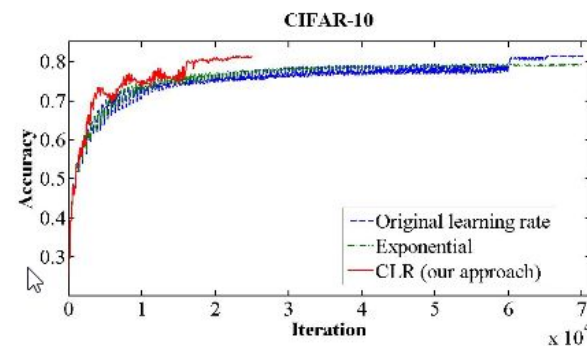
Learning Rate

- The amount that the weights are updated during training is referred to as the step size or the “learning rate.” Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.
- During training, the backpropagation of error estimates the amount of error for which the weights of a node in the network are responsible. Instead of updating the weight with the full amount, it is scaled by the learning rate. This means that a learning rate of 0.1, a traditionally common default value, would mean that weights in the network are updated $0.1 \times$ (estimated weight error) or 10% of the estimated weight error each time the weights are updated.
- Generally, a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train. At extremes, a learning rate that is too large will result in weight updates that will be too large and the performance of the model (such as its loss on the training dataset) will oscillate over training epochs. Oscillating performance is said to be caused by weights that diverge (are divergent). A learning rate that is too small may never converge or may get stuck on a suboptimal solution.
- The learning rate may, in fact, be the most important hyperparameter to configure for your model. Unfortunately, we cannot analytically calculate the optimal learning rate for a given model on a given dataset. Instead, a good (or good enough) learning rate must be discovered via trial and error.
- Diagnostic plots can be used to investigate how the learning rate impacts the rate of learning and learning dynamics of the model. An alternative approach is to perform a sensitivity analysis of the learning rate for the chosen model, also called a grid search. This can help to both highlight an order of magnitude where good learning rates may reside, as well as describe the relationship between learning rate and performance. When plotted, the results of such a sensitivity analysis often show a “U” shape, where loss decreases (performance improves) as the learning rate is decreased with a fixed number of training epochs to a point where loss sharply increases again because the model fails to converge.



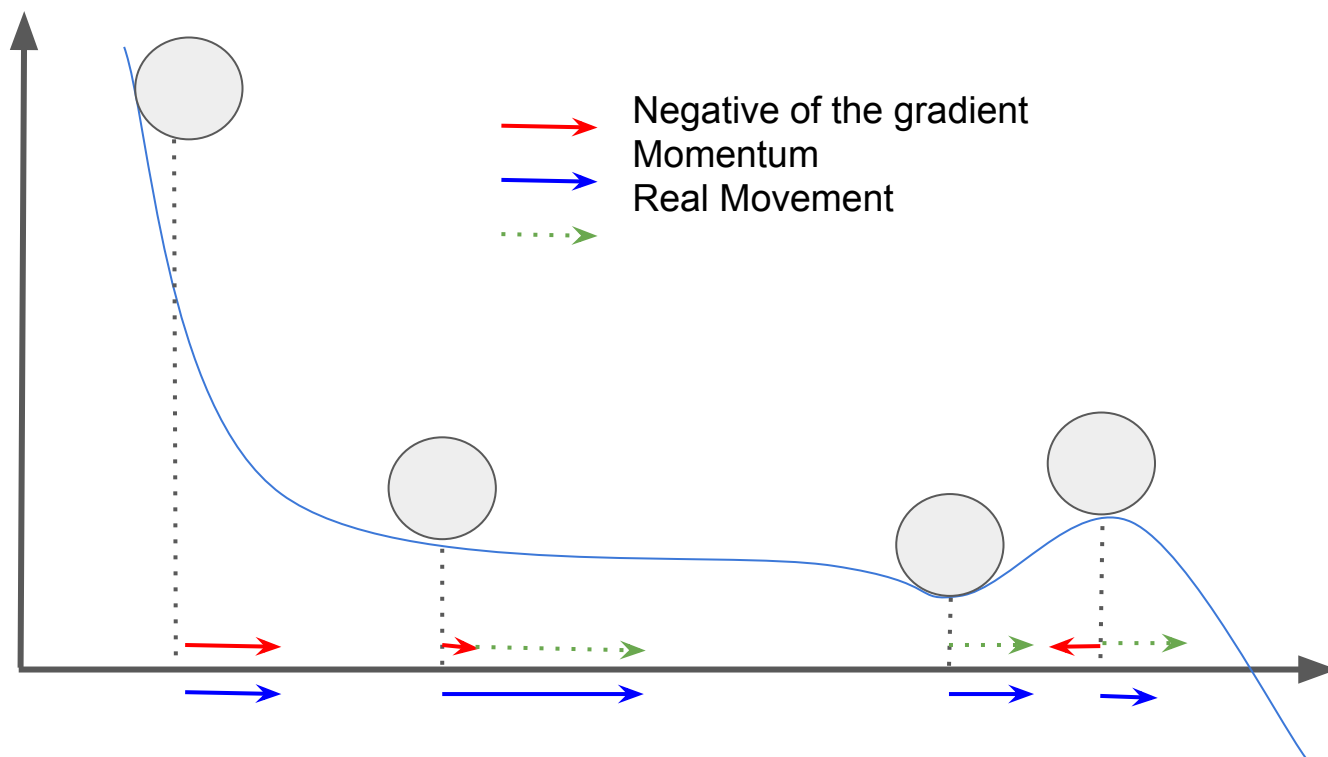
Adapting the Learning Rate

- An alternative to using a fixed learning rate is to instead vary the learning rate over the training process. The way in which the learning rate changes over time (training epochs) is referred to as the learning rate schedule or learning rate decay.
- One of the simplest learning rate schedules is to decrease the learning rate linearly from a large initial value to a small value. This allows large weight changes in the beginning of the learning process and small changes or fine-tuning towards the end of the learning process.
- The learning rate can be decayed to a small value close to zero. Alternately, the learning rate can be decayed over a fixed number of training epochs, then kept constant at a small value for the remaining training epochs to facilitate more time fine-tuning.
- The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response. This is called an adaptive learning rate.
- Perhaps the simplest implementation is to make the learning rate smaller once the performance of the model plateaus, such as by decreasing the learning rate by a factor of two or an order of magnitude. Alternately, the learning rate can be increased again if performance does not improve for a fixed number of training epochs.
- An adaptive learning rate method will generally outperform a model with a badly configured learning rate. Although no single method works best on all problems, there are three adaptive learning rate methods that have proven to be robust over many types of neural network architectures and problem types: Momentum, RMSProp and Adam.



Cyclical Learning Rates for Training
Neural Networks. Leslie N. Smith

MOMENTUM IDEA



Both are equivalent

$$\begin{aligned}
 V_t &= \beta V_{t-1} + (1 - \beta) \nabla_w L(W, X, y) & V_t &= \beta V_{t-1} + \alpha \nabla_w L(W, X, y) \\
 W &= W - \alpha V_t & W &= W - V_t
 \end{aligned}$$

RMSProp

$$\omega_t = \omega_{t-1} + \Delta\omega_t$$

$$\Delta\omega_t = -\frac{\mu}{\sqrt{v_t + \epsilon}} \cdot g_t$$

$$v_t = \rho \cdot v_{t-1} + (1 - \rho) \cdot g_t^2$$

v_t exponential average of squares of gradients
 μ initial learning rate
 g_t gradient at time t at ω

- RMSprop, or Root Mean Square Propagation was devised by the legendary Geoffrey Hinton, while suggesting a random idea during a Coursera class.
- RMSProp also tries to dampen the oscillations, but in a different way than momentum. Momentum accumulates the gradient of the past steps to determine the direction to go. RMS prop also takes away the need to adjust learning rate, and does it automatically. More so, RMSProp choses a different learning rate for each parameter.
- In RMS prop, each update is done according to the equations described below. This update is done separately for each parameter

In the third equation, we compute an exponential average of the square of the gradient. Since we do it separately for each parameter, gradient G_t here corresponds to the projection, or component of the gradient along the direction represented by the parameter we are updating.

To do that, we multiply the exponential average computed till the last update with a hyperparameter, represented by the greek symbol ρ . We then multiply the square of the current gradient with $(1 - \rho)$. We then add them together to get the exponential average till the current time step.

The reason why we use exponential average is it helps us weigh the more recent gradient updates more than the less recent ones. In fact, the name "exponential" comes from the fact that the weightage of previous terms falls exponentially (the most recent term is weighted as ρ , the next one as squared of ρ , then cube of ρ , and so on.)

The first equation is just the update step. The hyperparameter ρ is generally chosen to be 0.9, but you might have to tune it. The epsilon in equation 2, is to ensure that we do not end up dividing by zero and is generally chosen to be $1e-10$.

It's also to be noted that RMSProp implicitly performs simulated annealing. Suppose if we are heading towards the minima, and we want to slow down so as to not to overshoot the minima. RMSProp automatically will decrease the size of the gradient steps towards minima when the steps are too large (Large steps make us prone to overshooting)

Adam

$$\omega_t = \omega_{t-1} + \Delta\omega_t$$

$$\Delta\omega_t = -\frac{\mu \cdot v_t}{\sqrt{s_t + \epsilon}} \cdot g_t$$

$$v_t = \beta_1 \cdot v_{t-1} + (1 - \beta_1) \cdot g_t$$

$$s_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

v_t exponential average of gradients

s_t exponential average of squares of gradients

μ initial learning rate

β_1, β_2 hyperparameters

g_t gradient at time t at ω

- While momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations.
- **Adam or Adaptive Moment Optimization** algorithms combines the heuristics of both Momentum and RMSProp. Here are the update equations.
- Here, we compute the exponential average of the gradient as well as the squares of the gradient for each parameters (Eq 3, and Eq 4). To decide our learning step, we multiply our learning rate by average of the gradient (as was the case with momentum) and divide it by the root mean square of the exponential average of square of gradients (as was the case with momentum) in equation 2. Then, we add the update.
- The hyperparameter *beta1* is generally kept around 0.9 while *beta_2* is kept at 0.99. Epsilon is chosen to be 1e-10 generally.

Cost Functions

Some Regression Losses

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(\hat{y}_i - y_i)^2}{n}}$$

- The most classic measure
- Penalizes highly big mistakes
- Less interpretable

$$MAE = \sum_{i=1}^n \frac{|\hat{y}_i - y_i|}{n}$$

- Harder differentiability and convergence
- Penalizes less on higher mistakes
- Interpretable

$$MASE = \frac{\sum_{i=1}^n |\hat{y}_i - y_i|}{\frac{n}{n-1} \sum_{i=2}^n |y_n - y_{n-1}|}$$

- Scale invariant
- Symmetric
- Interpretable
- Harder differentiability and convergence

Classification Loss: Categorical Cross-Entropy

- Categorical Cross-Entropy

$$\mathcal{L}(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{ij} \log(p_{ij})$$

Where indexes i and j stand for each example and resp. class, the y s stand for the true labels and the p s stand for their assigned probabilities

On two classes it turns into the *easy-to-understand*, most common

$$\mathcal{L}(\theta) = -\frac{1}{n} \sum_{i=1}^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

When compared to accuracy, Cross-Entropy turns to be a more granular way to compute error closeness of a prediction, as it takes into account the closeness of a prediction.

Derivation also eases calculus compared with RMSE

Classifier 1

computed	targets	correct?
0.3 0.3 0.4	0 0 1 (democrat)	yes
0.3 0.4 0.3	0 1 0 (republican)	yes
0.1 0.2 0.7	1 0 0 (other)	no

Classifier 2

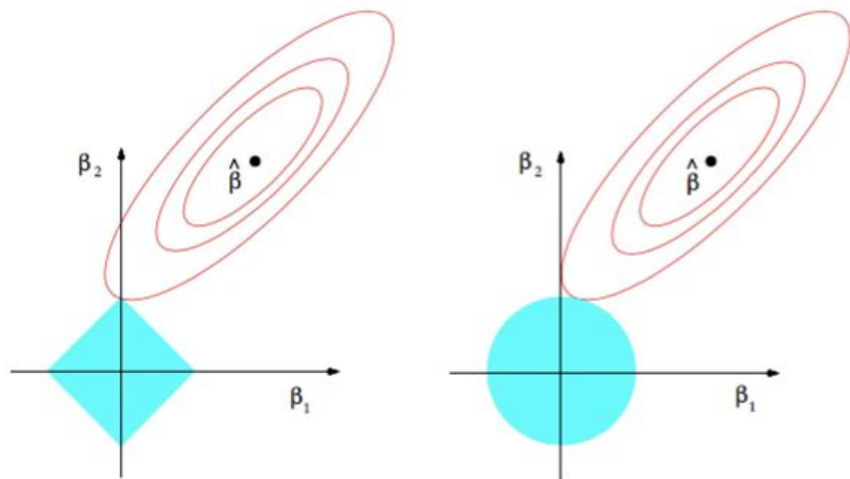
computed	targets	correct?
0.1 0.2 0.7	0 0 1 (democrat)	yes
0.1 0.7 0.2	0 1 0 (republican)	yes
0.3 0.4 0.3	1 0 0 (other)	no

$$\mathcal{L}(C_1) = 1.38; \mathcal{L}(C_2) = 0.64$$

$$Acc(C_1) = Acc(C_2) = 0.66$$

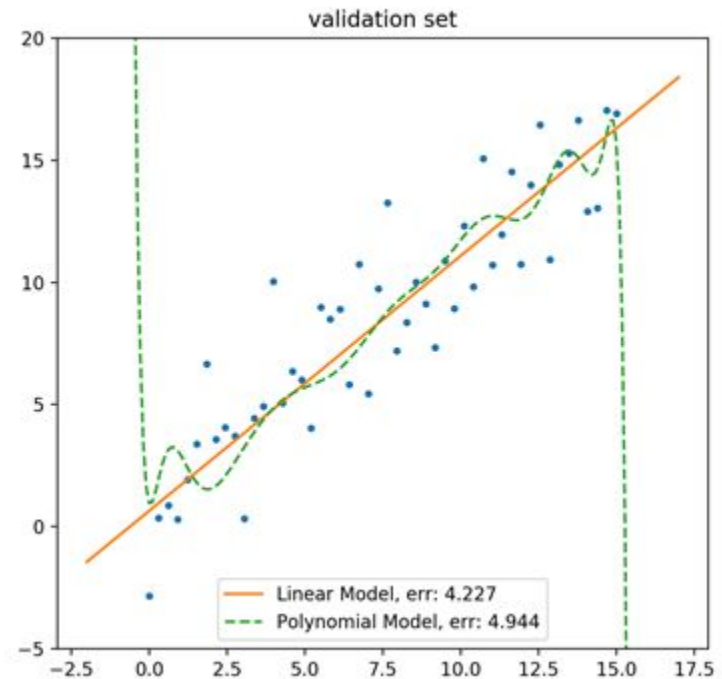
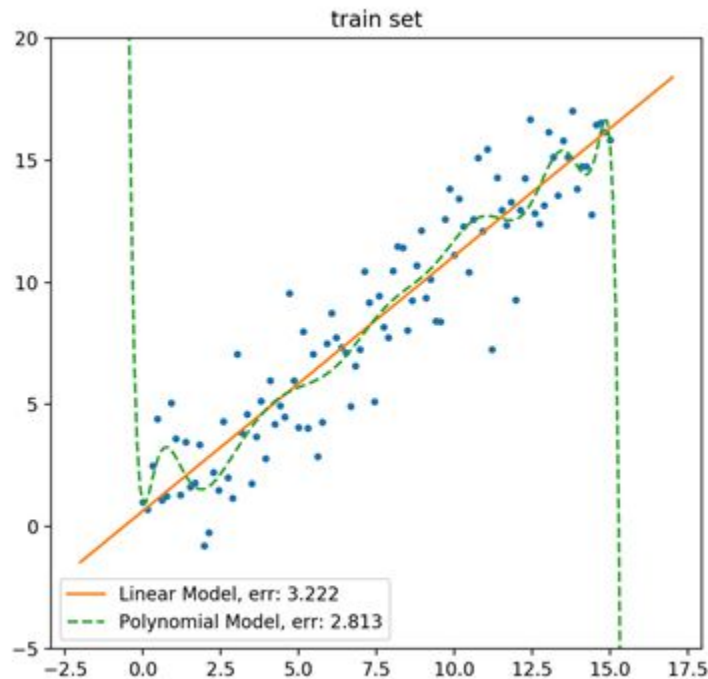
Regularization

Regularization Basics



- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.
- In general Machine Learning, the concept of regularization penalizes the coefficients.
- In deep learning, it actually penalizes the weight matrices of the nodes.
- Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero. This will result in a much simpler linear network and slight underfitting of the training data. Such a large value of the regularization coefficient is not that useful. We need to optimize the value of regularization coefficient in order to obtain a well-fitted model.

Overfitting Reloaded

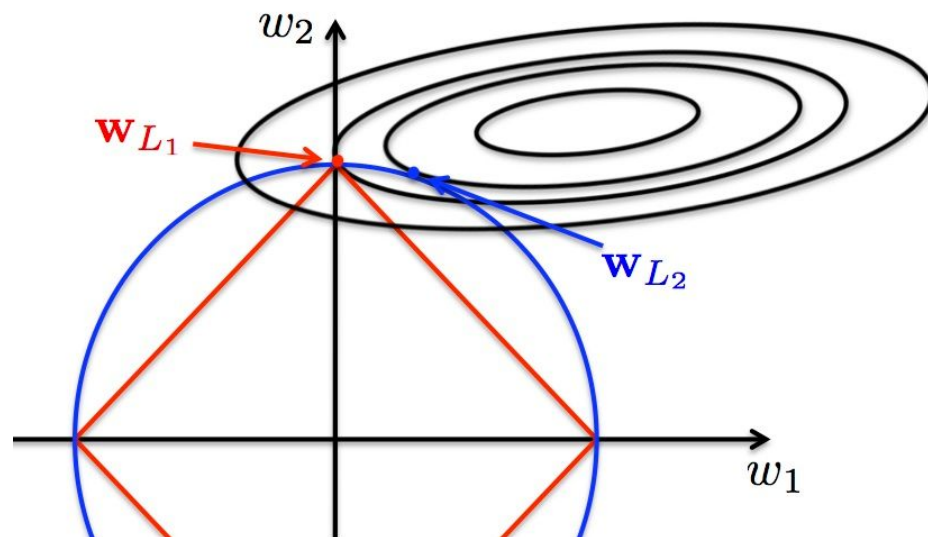


- In Deep Learning models tend to be very complex. With millions of parameters and sometimes too few data, overfitting is a risk
- Ideally with enough data, training, testing and validation data would have the same distribution. So, overfit to one of them would mean a less adaptive model, but a model that would behave good with unseen data.
- We have mentioned training, testing and validation data. Three data subsets are needed as our model fits and can overfit to our training data, but while we tune the model hyperparameters with the validation model performance, the model would also learn the validation data, and this way we need another testing dataset.

Weight Decay/lx regularization

- Add a penalty to the loss function: $\tilde{J}(\beta) = J(\beta) + \alpha F(\beta)$

- L2: $\|\beta\|_2^2 = \sum_i \beta_i^2$
 - Keep weights near zero.
 - Simplest one, differentiable.
- L1: $\|\beta\|_1 = \sum_i |\beta_i|$
 - Sparse results, feature selection.
 - Not differentiable, slower.

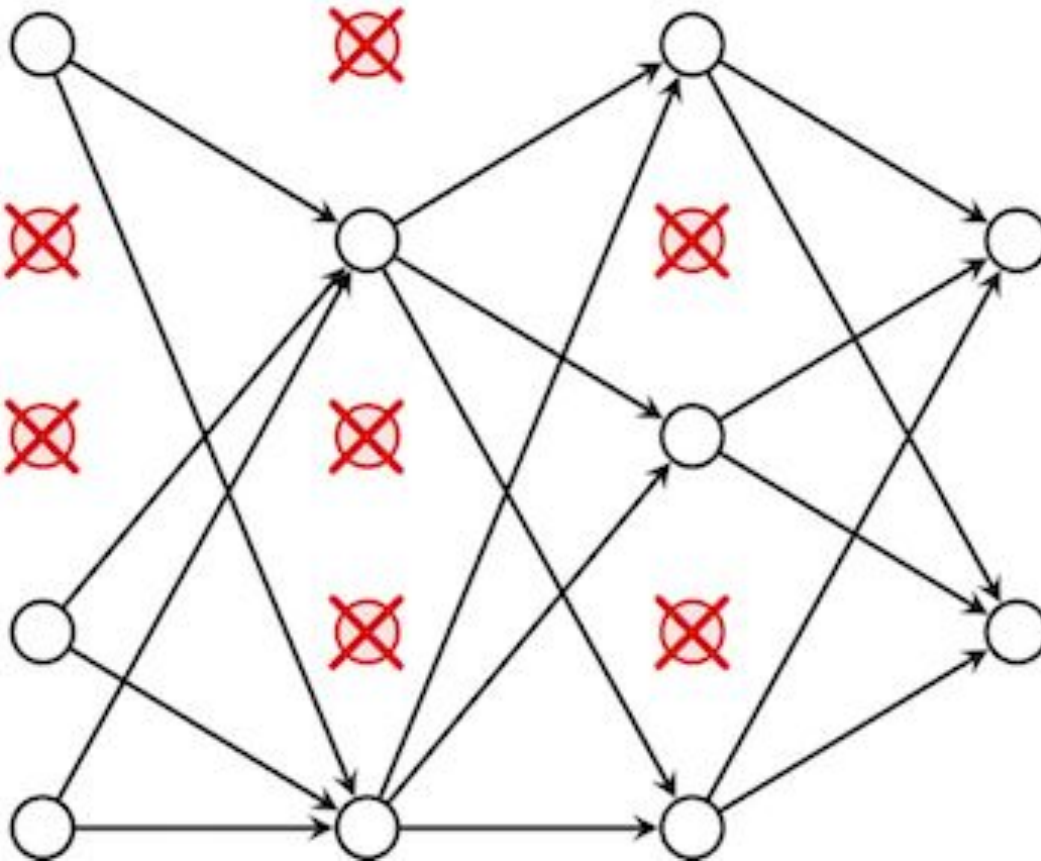


$$J_{\text{reg}}(\beta) = J(\beta) + \frac{\alpha}{2} \cdot \|\omega\|^2$$

$$\omega_t = \omega_{t-1} - \lambda \frac{\partial J_{\text{reg}}}{\partial \beta} = \omega_{t-1} - \lambda \frac{\partial J}{\partial \beta} - \lambda \cdot \alpha \cdot \omega$$

Both are equivalent, but when we speak about **L2 regularization**, it is considered like that in the first equation, while in the weight update one, we refer to it as **weight decay**

Dropout



- Randomly drop neurons (along with their connections) during training.
- Acts like adding noise.
- Very effective, computationally inexpensive.
- Ensemble of all sub-networks generated.
- Parametrized by the probability of each neuron to be dropped
- Not active on test/inference time

Batch Normalization

- Batch normalization is a technique where layers are inserted into typically a convolutional neural net that normalize the mean and scale of the per-channel activations of the previous layer. Depending on the architecture, this is usually somewhere between each nonlinear activation function and prior convolutional layers.
- Batch normalization was introduced by a 2015 paper, with the idea of stabilizing the distribution of layer activations over the course of training, reducing the instability of deeper neural nets to saturate or diverge. But exactly why it helps appears to still be a topic of research.
- Empirically for convolutional neural nets on some (but not all) problems, batch normalization stabilizes and accelerates the training while reducing the need to tune a variety of other hyperparameters to achieve reasonable performance.
- The key property is that batch norm layers make the neural net output approximately *invariant to the scale of the activations of the previous layers*. Any such scaling will simply be normalized away, except for the tiny effect of the ϵ in the denominator.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

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

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

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

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

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

Data Augmentation

Data Augmentation

- The performance of deep learning neural networks often improves with the amount of data available.
- Data augmentation is a technique to artificially create new training data from existing training data. This is done by applying domain-specific techniques to examples from the training data that create new and different training examples.
- Image data augmentation is perhaps the most well-known type of data augmentation and involves creating transformed versions of images in the training dataset that belong to the same class as the original image.
- Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more.
- The intent is to expand the training dataset with new, plausible examples. This means, variations of the training set images that are likely to be seen by the model. For example, a horizontal flip of a picture of a cat may make sense, because the photo could have been taken from the left or right. A vertical flip of the photo of a cat does not make sense and would probably not be appropriate given that the model is very unlikely to see a photo of an upside down cat.
- Image data augmentation is typically only applied to the training dataset, and not to the validation or test dataset. This is different from data preparation such as image resizing and pixel scaling; they must be performed consistently across all datasets that interact with the model.



Keras Generators

ImageDataGenerator class

[\[source\]](#)

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False, samplewise_center=False,
```

Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches).

Arguments

- **featurewise_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.
- **samplewise_center**: Boolean. Set each sample mean to 0.
- **featurewise_std_normalization**: Boolean. Divide inputs by std of the dataset, feature-wise.
- **samplewise_std_normalization**: Boolean. Divide each input by its std.
- **zca_epsilon**: epsilon for ZCA whitening. Default is 1e-6.
- **zca_whitening**: Boolean. Apply ZCA whitening.
- **rotation_range**: Int. Degree range for random rotations.
- **width_shift_range**: Float, 1-D array-like or int
 - float: fraction of total width, if < 1, or pixels if >= 1.
 - 1-D array-like: random elements from the array.
 - int: integer number of pixels from interval `(-width_shift_range, +width_shift_range)`
 - With `width_shift_range=2` possible values are integers `[-1, 0, +1]`, same as with `width_shift_range=[-1, 0, +1]`, while with `width_shift_range=1.0` possible values are floats in the interval `[-1.0, +1.0]`.
- **height_shift_range**: Float, 1-D array-like or int
 - float: fraction of total height, if < 1, or pixels if >= 1.
 - 1-D array-like: random elements from the array.
 - int: integer number of pixels from interval `(-height_shift_range, +height_shift_range)`
 - With `height_shift_range=2` possible values are integers `[-1, 0, +1]`, same as with `height_shift_range=[-1, 0, +1]`, while with `height_shift_range=1.0` possible values are floats in the interval `[-1.0, +1.0]`.
- **brightness_range**: Tuple or list of two floats. Range for picking a brightness shift value from.
- **shear_range**: Float. Shear Intensity (Shear angle in counter-clockwise direction in degrees)
- **zoom_range**: Float or [lower, upper]. Range for random zoom. If a float, `[lower, upper] = [1-zoom_range, 1+zoom_range]`.
- **channel_shift_range**: Float. Range for random channel shifts.
- **fill_mode**: One of {"constant", "nearest", "reflect" or "wrap"}. Default is 'nearest'. Points outside

- The Keras deep learning library provides the ability to use data augmentation automatically when training a model, generating augmented samples in real time and not storing them in memory.
- We have to keep in mind that in some cases, even the most state-of-the-art configuration won't have enough memory space to process the data the way we used to do it. This is achieved by using the `ImageDataGenerator` class.
- First, the class may be instantiated and the configuration for the types of data augmentation are specified by arguments to the class constructor.
- A range of techniques are supported, as well as pixel scaling methods. Some of the most used ones are the following:
 - Image shifts via the *width_shift_range* and *height_shift_range* arguments.
 - Image flips via the *horizontal_flip* and *vertical_flip* arguments.
 - Image rotations via the *rotation_range* argument
 - Image brightness via the *brightness_range* argument.
 - Image zoom via the *zoom_range* argument.

Conclusion

images



images



image_gen



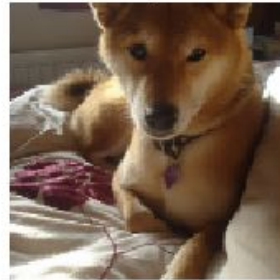
image_gen



image_gen



images



image_gen



images



images



An additional idea is to use a generative model to actually generate new image samples for training our model.

Most generative models take a lot of data to train. This makes the augmentation idea not feasible directly

However, some generative models are specialised on generating data *on different domains*. Our classifier may benefit from this if for example, we lack of dogs in the countryside.

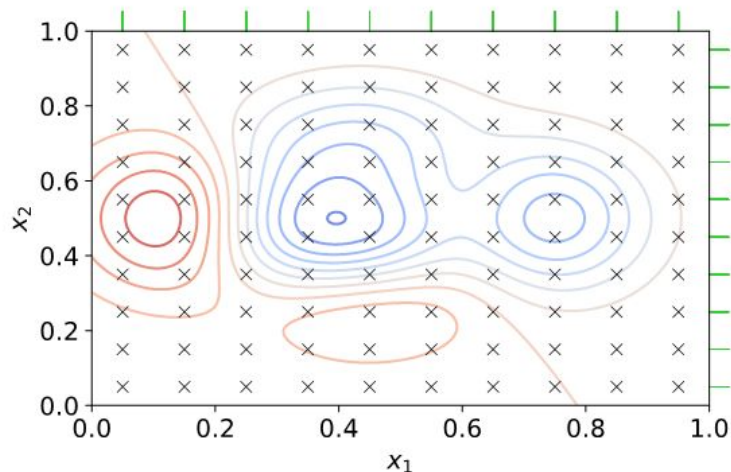
Hyperparameter Optimization

Hyperparameter Tuning. Manual Search

- *Hyperparameter tuning* refers to the automatic optimization of the hyper-parameters of a ML model. As we know, *hyperparameters* are all the parameters of a model which are not updated during the learning and are used to configure either the model or the algorithm used to lower the cost function. This idea can be pushed further to include the optimization algorithm as a hyper-parameter.
- Conceptually, hyper-parameter tuning is just an optimization loop on top of ML model learning to find the set of hyper-parameters leading to the lowest error on the validation set. Thus, a validation set has to be set apart, and a loss has to be defined.
- This outer-loop optimization is done on a ML system, and this has two implications. First, the evaluation of the function is very costly, as it entails learning a model (which can take hours if not days on a large-scale computing cluster) and evaluating its performance on a validation set.
- Secondly, the model is evaluated on a validation set, which is a substitute to get the expected performance on the true distribution of the data (which leads to the generalization error). The performance of the model is evaluated with some error, and thus there is no point in finding the true optimum with a high precision.
- As opposed to the optimization algorithm of the ML model, no gradient is computed, so the hyper-parameter optimization algorithm cannot rely on it to lower the validation error. Instead, it must blindly try a new configuration in the search space or make an educated guess of where the most interesting configuration might be. It is why these optimizers are called black-box optimizers
- One of the first techniques used to optimize model hyperparameters is **manual search**. This is the most intuitive, naïve way: we rely on the person behind the model for finding the optimal values.
- Manual search is too dependent on past results of another models and will not deterministically find an optimum value for the hyperparameter configuration most of the times. It depends on expertise and is biased through previous trainings. Nevertheless, it brings fair enough models most of the times for people with high expertise.

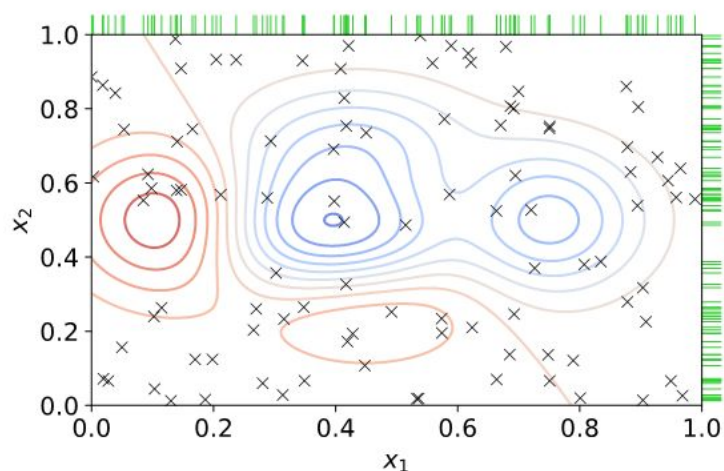


Grid Search



- In Grid Search, we set up a grid of hyperparameters and train/test our model on each of the possible combinations.
- The search space of each hyper-parameter is discretized, and the total search space is discretized as the Cartesian products of them. Then, the algorithm launches a learning for each of the hyper-parameter configurations, and selects the best at the end.
- As we will try each one of the possible configurations in our grid, it will deterministically find the optimum configuration within this discrete set of values... If these values are in fact discrete.
- It is an *embarrassingly* parallel problem (provided one has the computing power needed to train several models at the same time) but suffers from the curse of dimensionality (the number of configurations to try is exponential with regards to the number of hyper-parameters to be optimized).
- As an example, if we have to configure two optimizers, 10 different neurons on the first layer, 4 learning rates, 4 different regularizations, 4 hyperparameter values for those (e.g: dropout percentage) and 10 different amounts of neurons for the second layer, this means 128000 combinations ($2 \times 10 \times 4 \times 4 \times 4 \times 10$) which will be used during training. Is this optimal?

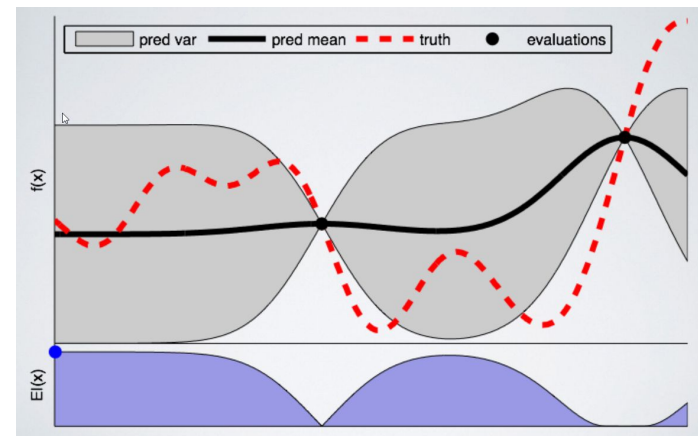
Random Search



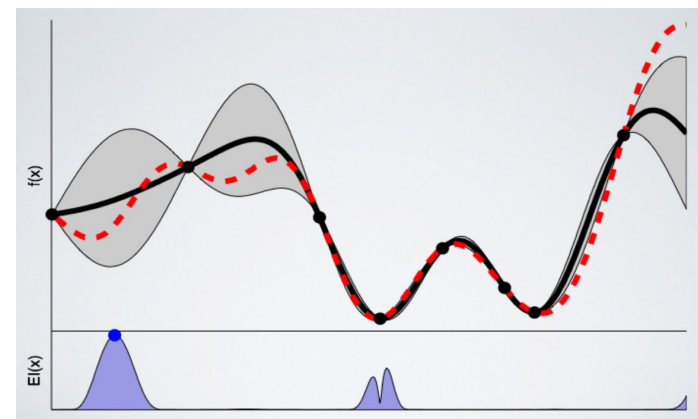
- Random Search is a variation of the previous algorithm, which randomly samples the search space instead of discretizing it with a Cartesian grid. The algorithm has no end. Instead a time budget has to be specified (in other words, a number of trials). This algorithm suffers likewise from the curse of dimensionality to reach a preset fixed sampling density.
- One of the advantages of random search is that if two hyper-parameters are little correlated, random search enables to find more precisely the optima of each parameter.
- Random search has proven to be particularly effective, especially if the search space is not cubic, i.e. if some hyper-parameters are given a much greater range of variation than others.
- Using purely random sampling for picking the next hyper-parameter sample can give a non-uniform sampling density of the search space.
- Random search and grid search are an attractive first option for optimization. They are very easy to code, can be run in parallel and they do not need any form of tuning. Their drawback is that there is no guarantee of finding a local minimum to some precision except if the search space is thoroughly sampled. This does not pose any problem if the model is very fast to run and the number of hyper-parameters is low.
- If the model takes a significant time to run using a lot of computational resources, random or grid search are inefficient as they do not use the information gained by all the previous tries.

Bayesian Optimization

- Bayesian approaches, in contrast to random or grid search, keep track of past evaluation results which they use to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function $P(\text{score}|\text{hyperparameters})$. This model is called a “surrogate” for the objective function and is represented as $p(y | x)$. The surrogate is much easier to optimize than the objective function and Bayesian methods work by finding the next set of hyperparameters to evaluate on the actual objective function by selecting hyperparameters that perform best on the surrogate function. In other words:
 - 1. Build a surrogate probability model of the objective function
 - 2. Find the hyperparameters that perform best on the surrogate
 - 3. Apply these hyperparameters to the true objective function
 - 4. Update the surrogate model incorporating the new results
 - 5. Repeat steps 2–4 until max iterations or time is reached
- The aim of Bayesian reasoning is to become “less wrong” with more data which these approaches do by continually updating the surrogate probability model after each evaluation of the objective function.
- At a high-level, Bayesian optimization methods are efficient because they choose the next hyperparameters in an *informed manner*. The basic idea is: spend a little more time selecting the next hyperparameters in order to make fewer calls to the objective function. In practice, the time spent selecting the next hyperparameters is inconsequential compared to the time spent in the objective function. By evaluating hyperparameters that appear more promising from past results, Bayesian methods can find better model settings than random search in fewer iterations.
- Bayesian model-based methods can find better hyperparameters in less time because they reason about the best set of hyperparameters to evaluate based on past trials.



Initial estimate of the surrogate model — in black with associated uncertainty in gray — after two evaluations. Clearly, the surrogate model is a poor approximation of the actual objective function in red.



Surrogate function after 8 evaluations. Now the surrogate almost exactly matches the true function. Therefore, if the algorithm selects the hyperparameters that maximize the surrogate, they will likely yield very good results on the true evaluation function.

