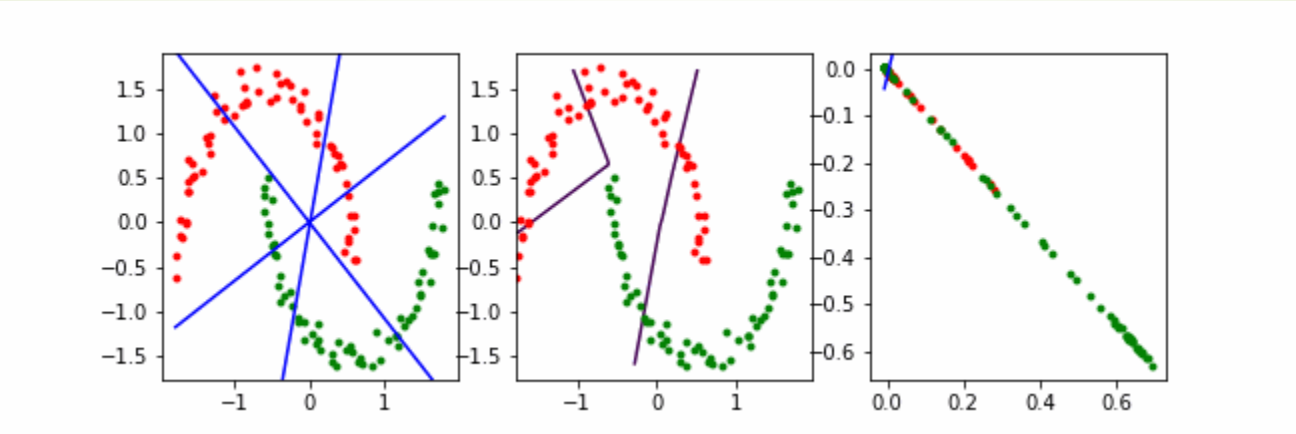
Module 3

Regularization and Optimization

Issues with Neural Network



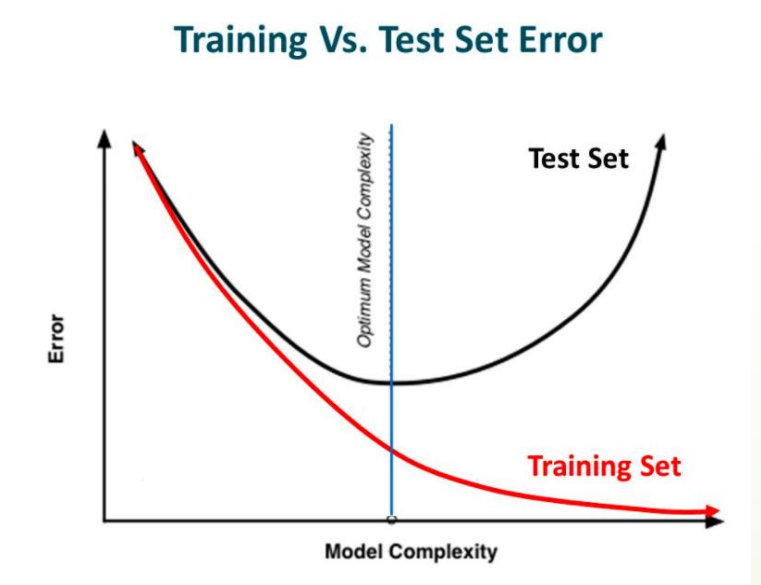
Isn't fitting a function to our data our end goal?

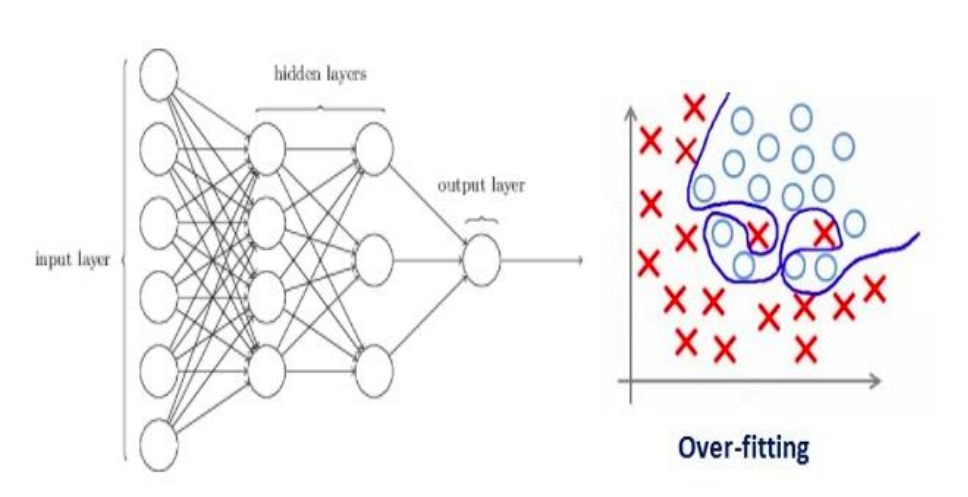
When a neural network overfits on the training dataset,

it learns an overly complex representation that models the training dataset too well.

As a result, it performs exceptionally well on the training dataset but generalizes poorly to unseen test data.

Issues with Neural Network





Bias and Variance?

Issues with Neural Network

- How well a model performs on training/evaluation datasets will define its characteristics

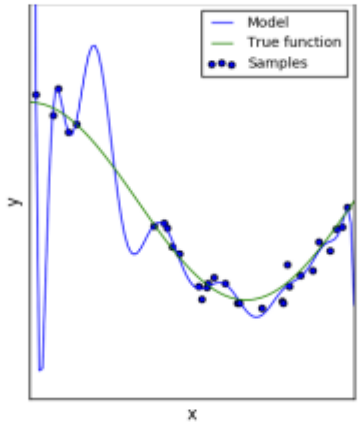
	Underfit	Overfit	Good Fit
Training Dataset	Poor	Very Good	Good
Evaluation Dataset	Very Poor	Poor	Good

- What causes underfit?
  - Model capacity is too small to fit the training dataset as well as generalize to new dataset.
  - High bias, low variance
- Solution
  - Increase the capacity of the model
  - Examples:
    - Increase number of layers, neurons in each layer, etc.
- Result:
  - Lower Bias
  - ***Underfit → Good Fit?***

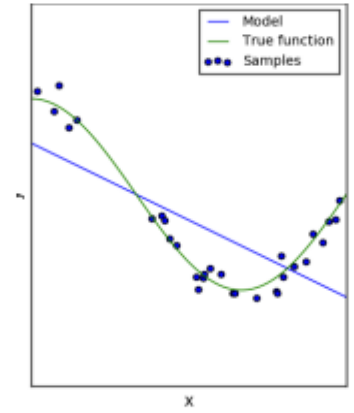
- *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.* [4]

Issues with Neural Network

Regularization—prevent overfitting



Optimization—overcome underfitting



Issues with Neural Network

Training Error	Valid Error	Cause	Solution
High	High	High bias	- Increase model complexity - Train for more epochs
Low	High	High variance	- Add more training data (e.g., **dataset augmentation**) - Use **regularization** - User **early stopping** (train less)
Low	Low	Perfect tradeoff	- You are done!

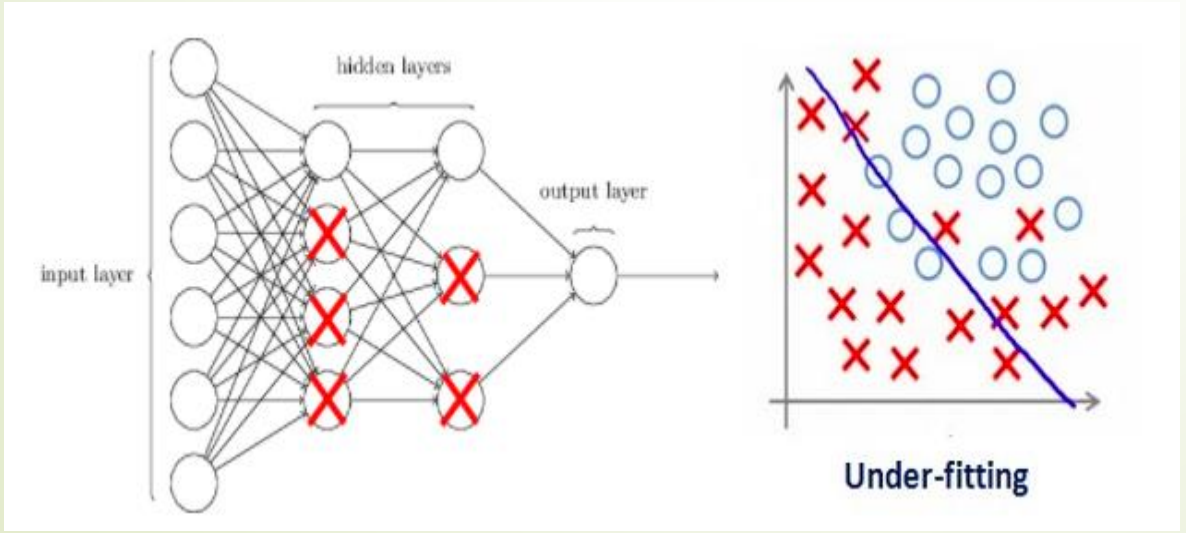
Regularization

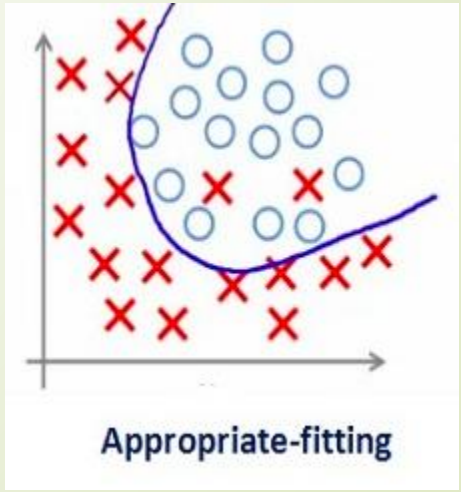
- ✓ Prevent overfitting and improve the generalization of neural networks.
- ✓ It involves adding a penalty term to the loss function during training.

Regularization penalizes the coefficients. In deep learning, it actually penalizes the weight matrices of the nodes.

- Weight Penalties → Smaller Weights → Simpler Model → Less Overfit

We need to optimize the value of the regularization coefficient in order to obtain a well-fitted model





Regularization

Regularization penalizes the coefficients. In deep learning, it actually penalizes the weight matrices of the nodes.

- **Early Stopping**
- **L1/L2**
- **Batch Norm**
- **Dropout**
- Data Augmentation
- Layer Norm
- Weight Norm

Different Regularization Techniques

L2 & L1 regularization

Cost function = Loss (say, binary cross entropy) + Regularization term

In L2, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$$

$$\min_\theta J(\theta)$$

In L1, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)
```

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

Different Regularization Techniques

L2 & L1 regularization

Cost function = Loss (say, binary cross entropy) + Regularization term

L1 Regularisation	L2 Regularisation
Sum of absolute value of weights	Sum of square of weights
Sparse solution is the outcome	Non-sparse (more segregated) solution
Multiple solutions are possible	Only one solution
Built-in feature selection in the penalty term	No specific feature selection mechanism
Robust to outliers	Not robots to outliers due to square term
Used in datasets with sparse features	Used in dataset with complex features

Different Regularization Techniques

L2 & L1 regularization

Cost function = Loss (say, binary cross entropy) + Regularization term

L1 regularization makes some coefficients zero, meaning the model will ignore those features. Ignoring the least important features helps emphasize the model's essential features.

Where lamda controls the strength of regularization, and w are the model's weights (coefficients).

A regularization term is added to the cost function of the linear regression, which keeps the magnitude of the model’s weights (coefficients) as small as possible. The L2 regularization technique tries to keep the model’s weights close to zero, but not zero, which means each feature should have a low impact on the output while the model's accuracy should be as high as possible.

Different Regularization Techniques

1.Lasso Regularization – L1 Regularization
2.Ridge Regularization – L2 Regularization
3.Elastic Net Regularization – L1 and L2 Regularization

Elastic Net Regression

This model is a combination of L1 as well as L2 regularization. That implies that we add the absolute norm of the weights as well as the squared measure of the weights. With the help of an extra hyperparameter that controls the ratio of the L1 and L2 regularization.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\left((1-\alpha)\sum_{i=1}^{m}|w_i| + \alpha\sum_{i=1}^{m}w_i^2\right)$$

1. A hyperparameter “alpha” is provided to assign how much weight is given to each of the L1 and L2 penalties. Alpha is a value between 0 and 1 and is used to weight the contribution of the L1 penalty and one minus the alpha value is used to weight the L2 penalty.

Different Regularization Techniques

Lasso Regularization/L1-norm Regularization

$$Cost\ (W) = RSS(W) + \lambda * (sum\ of\ absolute\ value\ of\ weights)$$

$$= \sum_{i=1}^{N} \left\{ y_i - \sum_{j=0}^{M} w_j\, x_{ij} \right\}^2 + \lambda \sum_{j=0}^{M} |w_j|$$

Ridge Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2}_{\text{Loss function}} + \underbrace{\lambda \sum_{j=0}^{M} W_j^2}_{\text{Regularization Term}}$$

Different Regularization Techniques

Lasso Regularization

```
# Create a Lasso regression model with regularization parameter alpha
lasso_model = Lasso(alpha=0.1)
# Fit the model to the training data
lasso_model.fit(X_train, y_train)
# Predict using the trained model
y_pred = lasso_model.predict(X_test)
```

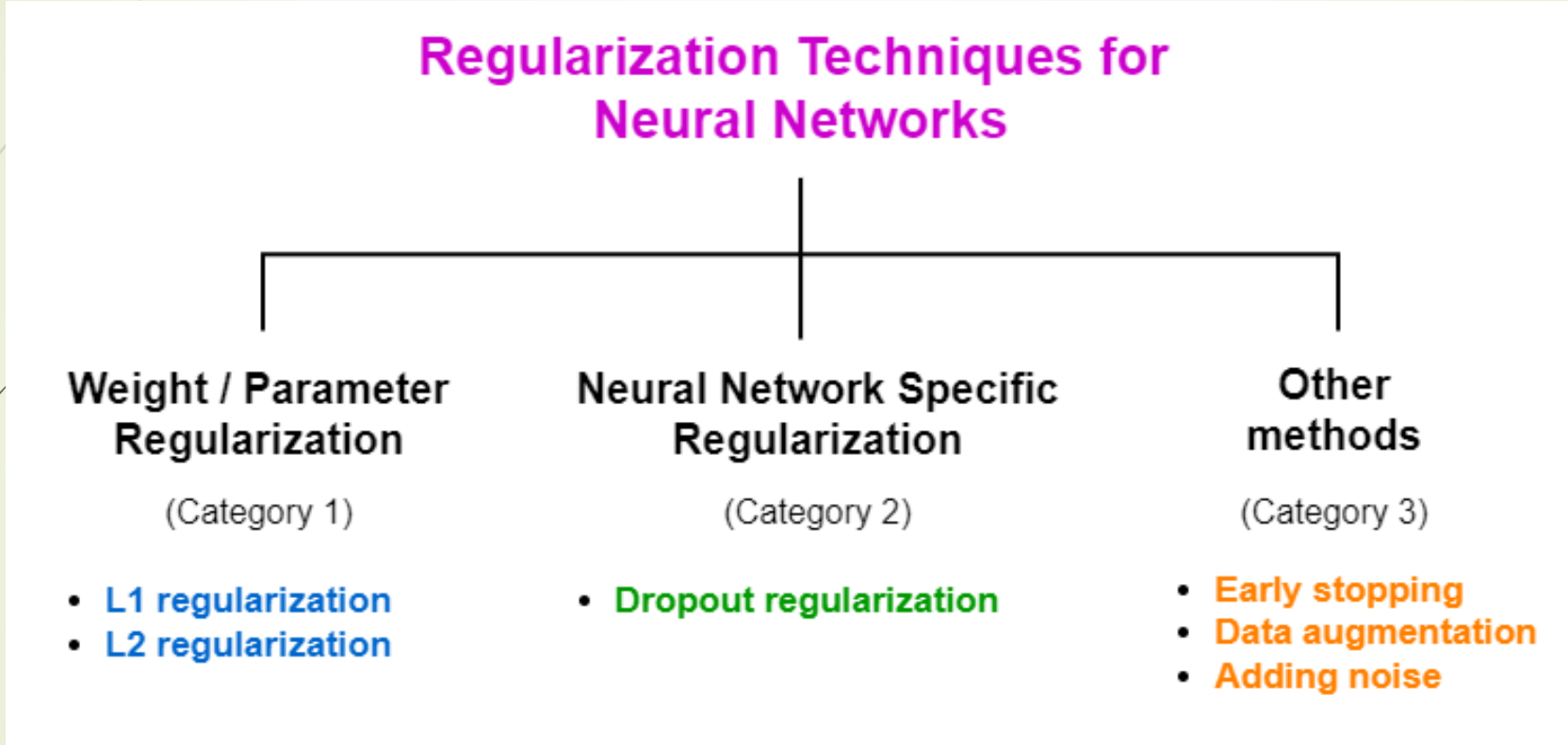
Ridge Regularization

```
from sklearn.linear_model import Ridge
# Create a Ridge regression model with regularization parameter alpha
ridge_model = Ridge(alpha=0.5)
# Fit the model to the training data
ridge_model.fit(X_train, y_train)
# Predict using the trained model
y_pred = ridge_model.predict(X_test)
```

Elastic Net Regularization

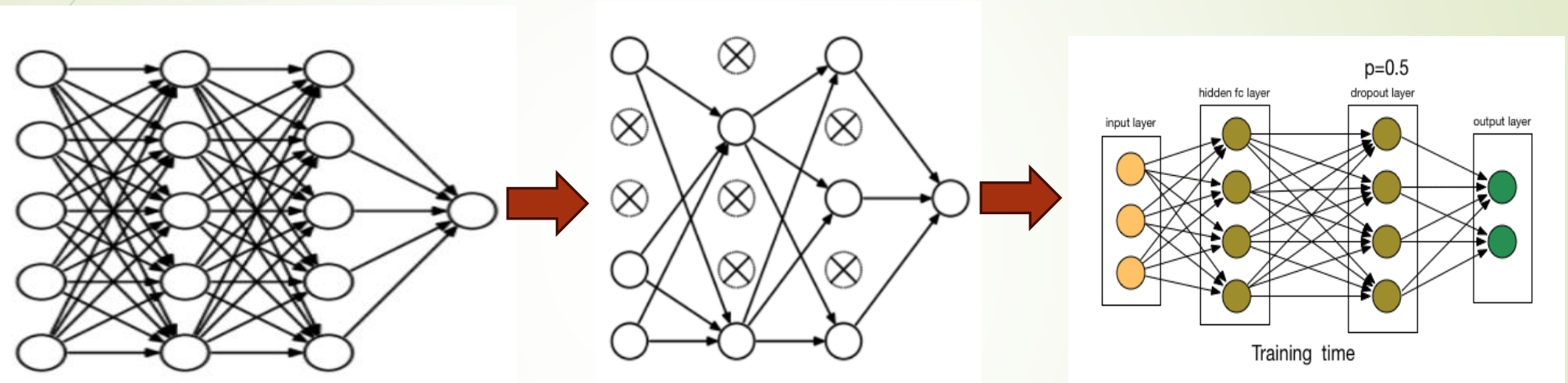
```
from sklearn.linear_model import ElasticNet
# Create an Elastic Net model with regularization parameters alpha and
l1_ratio
elasticnet_model = ElasticNet(alpha=0.1, l1_ratio=0.5)
# Fit the model to the training data
elasticnet_model.fit(X_train, y_train)
# Predict using the trained model
y_pred = elasticnet_model.predict(X_test)
```

Different Regularization Techniques



Different Regularization Techniques

Dropout



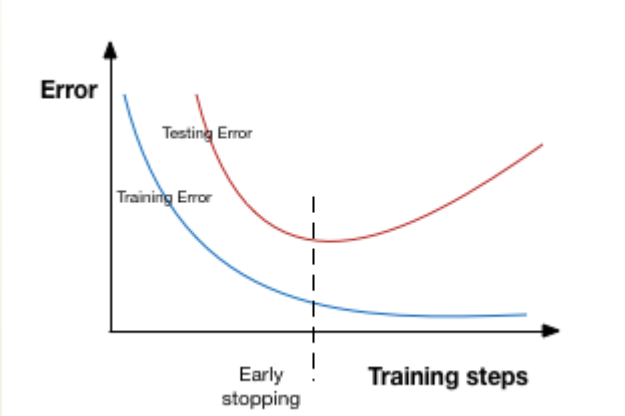

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])
```

Different Regularization Techniques

Data Augmentation


```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

Early stopping

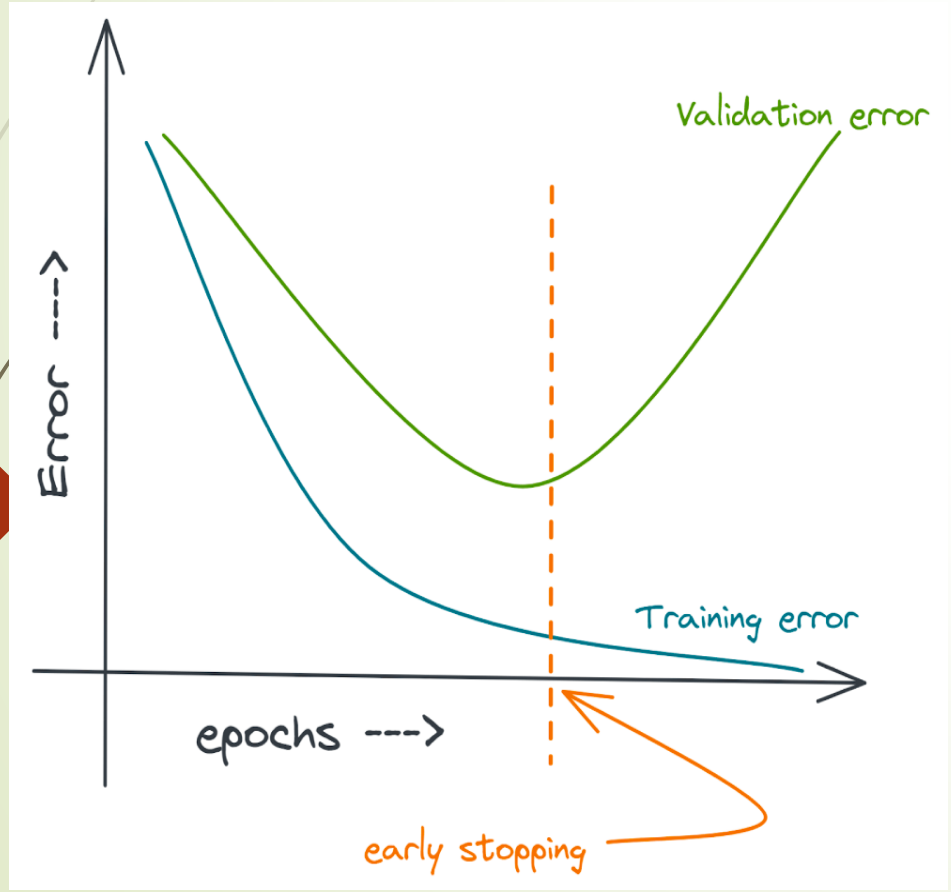


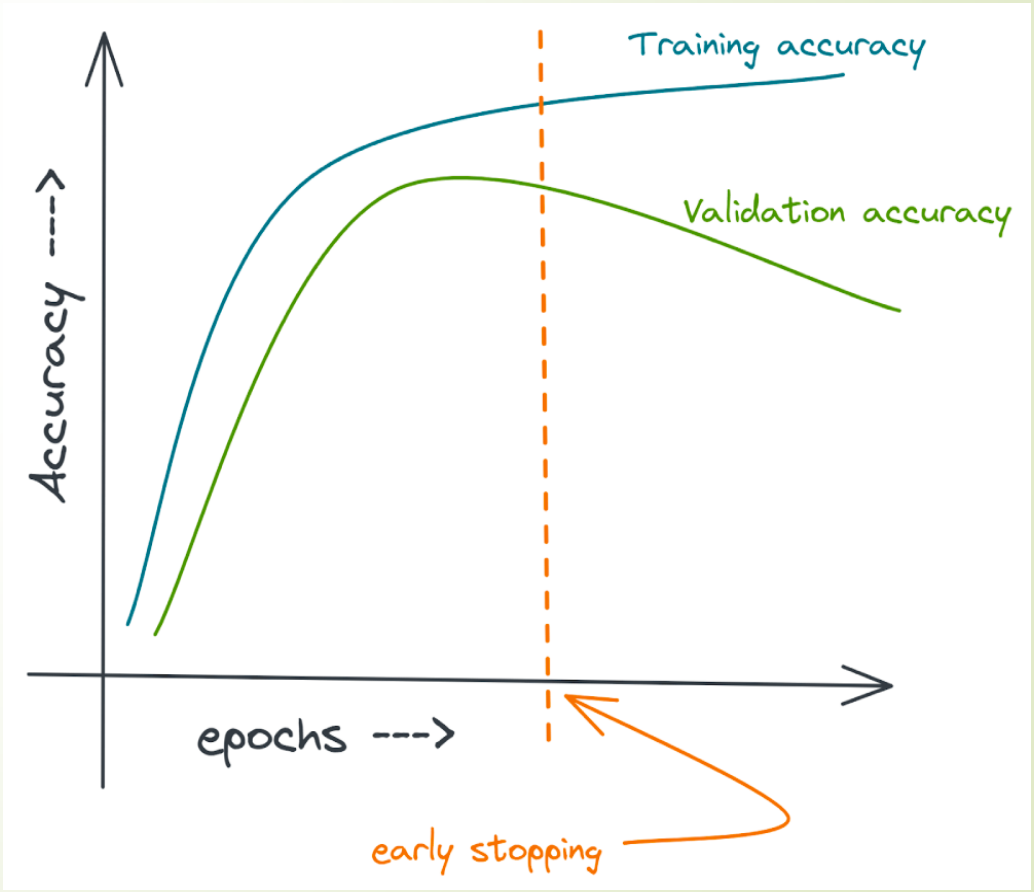

```
from keras.callbacks import EarlyStopping

EarlyStopping(monitor='val_err', patience=5)
```

Different Regularization Techniques

Early stopping is one of the simplest and most intuitive regularization techniques. It involves stopping the training of the neural network at an earlier epoch; hence the name early stopping.

Early stopping

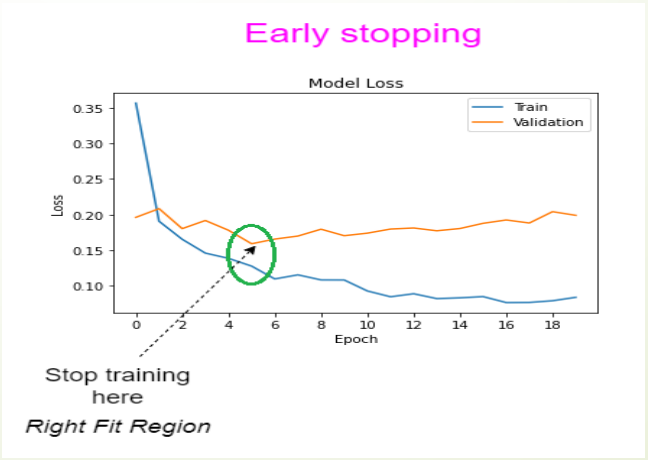




Different Regularization Techniques

Early stopping is one of the simplest and most intuitive regularization techniques. It involves stopping the training of the neural network at an earlier epoch; hence the name early stopping.

- Pros
  - Very simple
  - Highly recommend to use for all training along with other techniques
  - Keras Implementation has option to save BEST_WEIGHT
    - https://keras.io/callbacks/
    - Callback during training
- Cons
  - May not work so well



Different Regularization Techniques

Early stopping

Too little training will mean that the model will underfit the train and the test sets. Too much training will mean that the model will overfit the training dataset and have poor performance on the test set.

A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called early stopping.

1. The Problem of Training Just Enough
2. Stop Training When Generalization Error Increases
3. How to Stop Training Early
4. Examples of Early Stopping
5. Tips for Early Stopping

Different Regularization Techniques

Early stopping

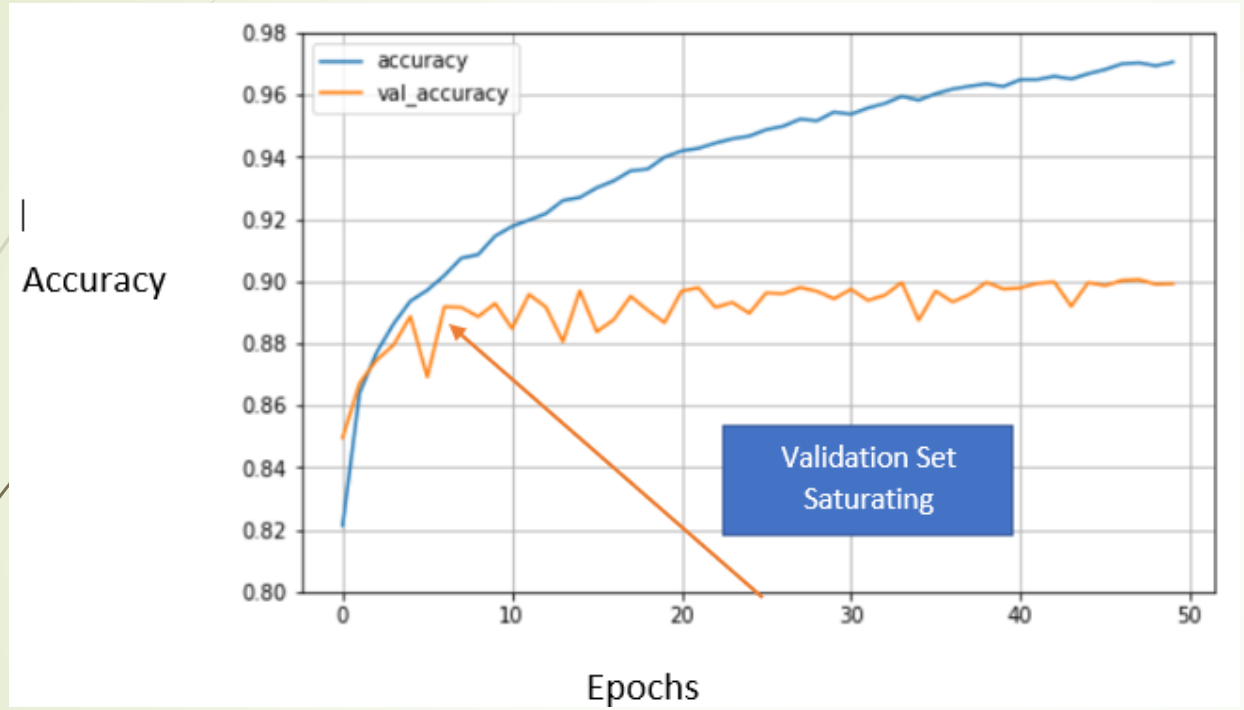
All standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting. While the network seems to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases.

One approach to solving this problem is to treat the number of training epochs as a hyperparameter and train the model multiple times with different values, then select the number of epochs that result in the best performance on the train or a holdout test dataset.

The downside of this approach is that it requires multiple models to be trained and discarded. This can be computationally inefficient and time-consuming, especially for large models trained on large datasets over days or weeks.

Different Regularization Techniques

Early stopping



Callback APIs:

we monitor what is happening when the model is getting trained and how do keep track?

Different Regularization Techniques

Callback APIs:

Some important parameters of the Early Stopping Callback:

- **monitor:** Quantity to be monitored. by default, it is validation loss
- **min_delta:** Minimum change in the monitored quantity to qualify as improvement
- **patience:** Number of epochs with no improvement after which training will be stopped.
- **mode:** One of {"auto", "min", "max"}. Is it a maximization problem or a minimization problem, we maximize accuracy, minimize loss.
- **restore_best_weights:** Whether to use best models weight or the last epoch weight

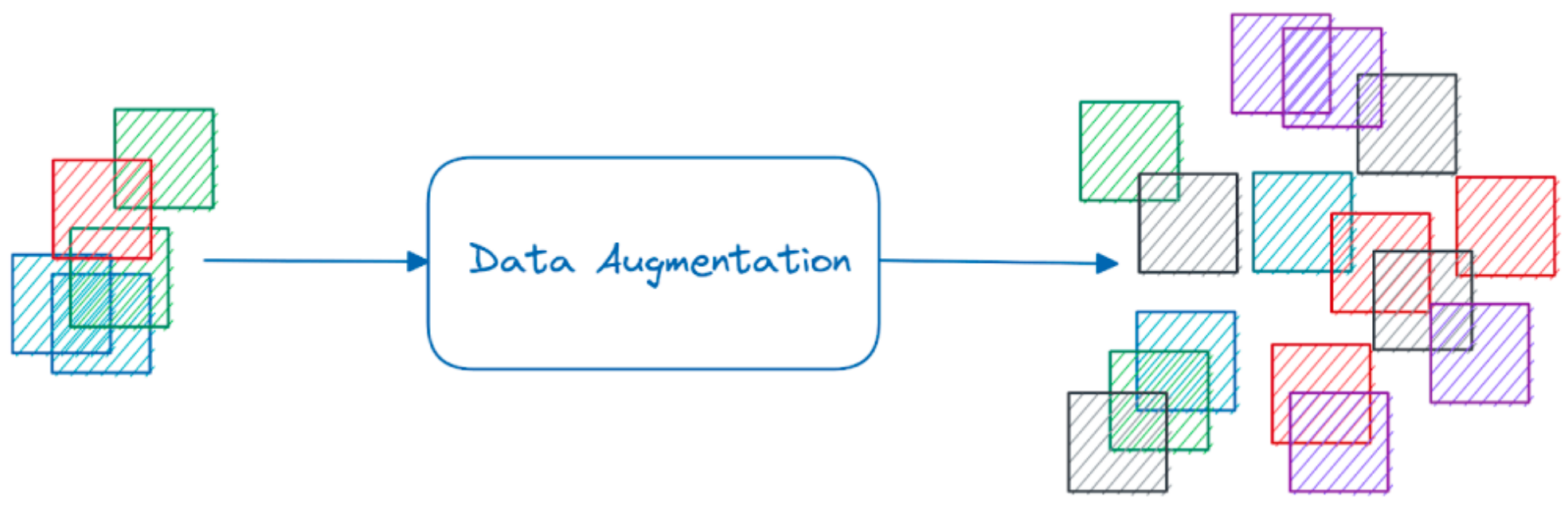
```
callback = tf.keras.callbacks.EarlyStopping(patience=4,
restore_best_weights=True)
history1 = model2.fit(trn_images, trn_labels,
epochs=50,validation_data=(valid_images, valid_labels),callbacks=
[callback])
```

Without early stopping, the model runs for all 50 epochs and we get a validation accuracy of 88.8%, with early stopping this runs for 15 epochs and the test set accuracy is 88.1%.

Different Regularization Techniques

Early stopping is one of the simplest and most intuitive regularization techniques. It involves stopping the training of the neural network at an earlier epoch; hence the name early stopping.

Data Augmentation



Optimization

- In deep learning, optimizers are algorithms that adjust the model's parameters during training to minimize a loss function.
- They enable neural networks to learn from data by iteratively updating weights and biases. Common optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop.
- Each optimizer has specific update rules, learning rates, and momentum to find optimal model parameters for improved performance.
- An optimizer is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates.
- Thus, it helps in reducing the overall loss and improving accuracy. The problem of choosing the right weights for the model is a daunting task, as a deep learning model generally consists of millions of parameters.

Optimizer Algorithms

First-order optimization algorithm

first-order methods use the first derivatives of the function to minimize.

1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. **RMSprop**
6. **Adam**
7. Adamax
8. Nadam
9. AMSGrad

Summary of GD, SGD, Minibatch SGD

Stochastic Gradient Descent [SGD]

Update rule

$$w_{new} = w_{old} - \eta \left(\frac{\partial L}{\partial w}\right)_{old}$$

* If $\frac{\partial L}{\partial w}$ is calculated using all the n points in D [training set] → GD

* If $\frac{\partial L}{\partial w}$ is calculated using only 1 point → SGD

* If $\frac{\partial L}{\partial w}$ is calculated using a random subset of K points in D → mini batch SGD

SGD with momentum

Minibatch SGD =

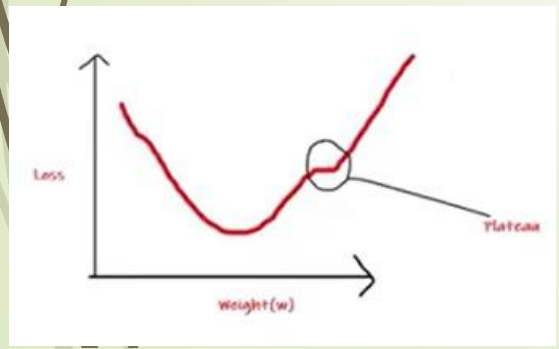
$$w_{new} = w_{old} - \eta \left(\frac{\partial L}{\partial w}\right)_{old}$$

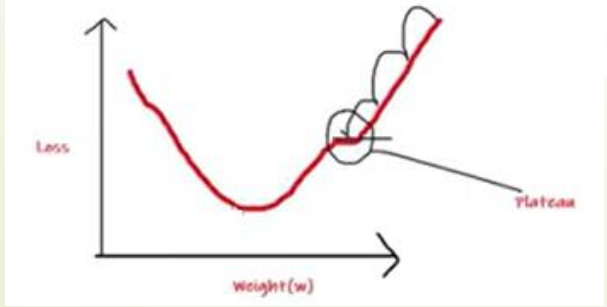
$$w_{new} = w_{old} - \eta v_t$$

$$v_t = \beta v_{t-1} + (1-\beta)\frac{\partial L}{\partial w}$$

↳ β is the hyperparameter of momentum

$$w_{new} = w_{old} - \eta\left[\beta v_{t-1} + (1-\beta)\frac{\partial L}{\partial w}\right]$$





If $\beta = 0$, then $v_t = 0 + \frac{\partial L}{\partial w} \Rightarrow v_t = \frac{\partial L}{\partial w}$

$$\boxed{w_{new} = w_{old} - \eta \frac{\partial L}{\partial w}}$$

Typically people uses $\beta = 0.9$

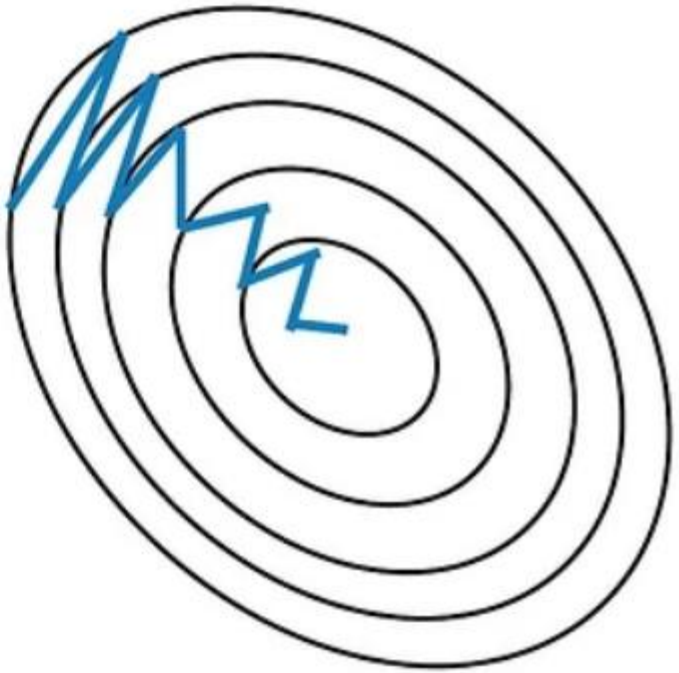
$$v_t = 0.9\, v_{t-1} + (1-0.9)\frac{\partial L}{\partial w}$$

$$= 0.9 v_{t-1} + 0.1\frac{\partial L}{\partial w}$$

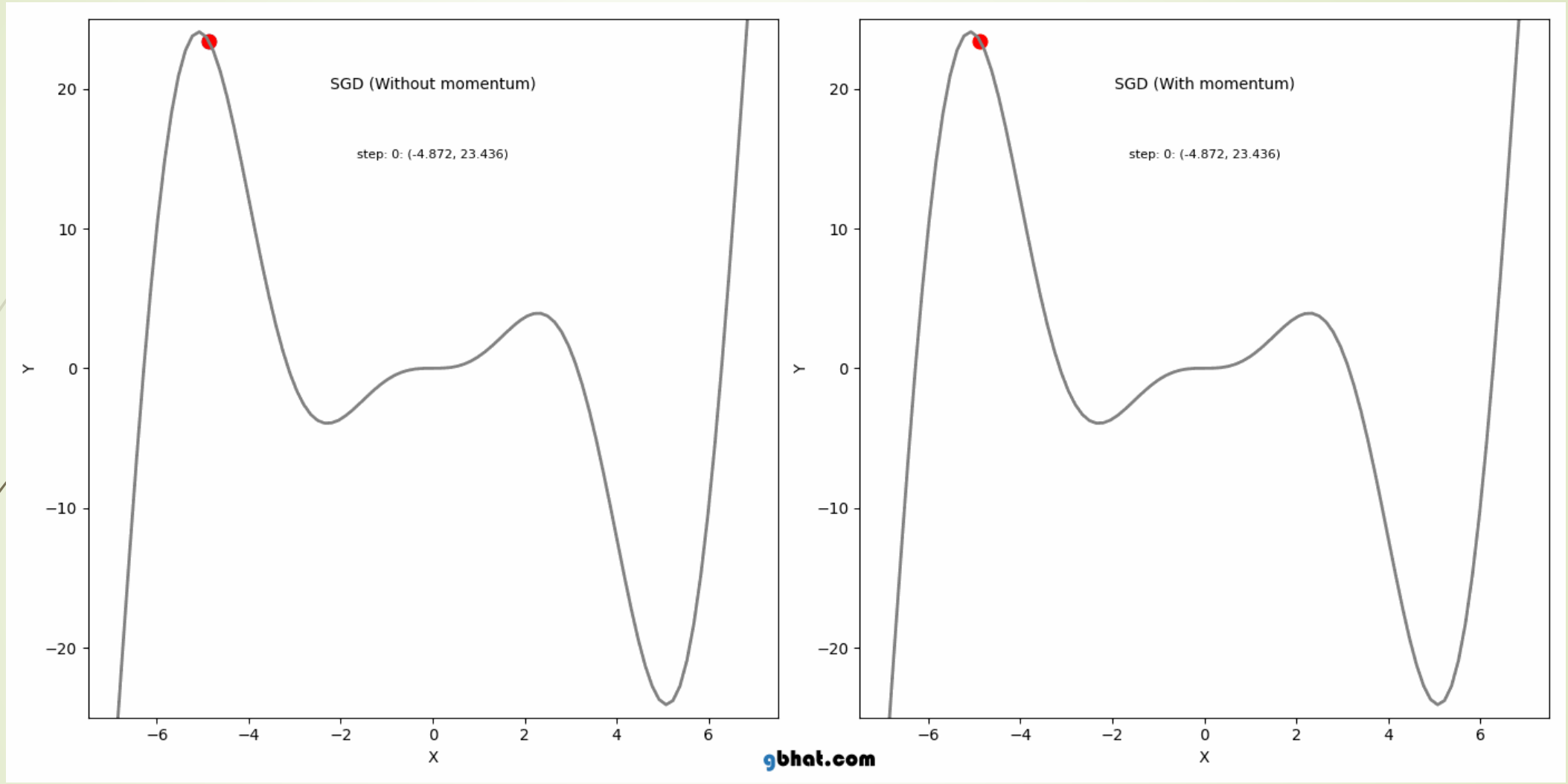
$$\therefore w_{new} = w_{old} - \eta\left[0.9 v_{t-1} + 0.1\frac{\partial L}{\partial w}\right]$$

↳ momentum ↳ Gradient

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum


SGD (Without momentum)
step: 0: (-4.872, 23.436)
SGD (With momentum)
step: 0: (-4.872, 23.436)
20
10
0
−10
−20
Y
−6
−4
−2
0
2
4
6
X
9bhat.com

Sparse dataset example

Users					Movies						Target
A	B	C	D	E	Parasite	Joker	Avengers	Spotlight	The Great Beauty	There will be blood	Rating
1	0	0	0	0	1	0	0	0	0	0	5
1	0	0	0	0	0	1	0	0	0	0	4
1	0	0	0	0	0	0	1	0	0	0	4
0	1	0	0	0	1	0	0	0	1	0	2
0	1	0	0	0	0	0	0	1	0	0	4
0	1	0	0	0	0	0	0	0	1	0	3
0	0	1	0	0	0	0	1	0	0	0	5
0	0	0	1	0	0	0	0	0	0	1	4
0	0	0	0	1	0	0	1	0	0	0	4

Adaptive Gradient (AdaGrad)

In SGD and mini-batch SGD, the value of η used to be the same for each weight, or say for each parameter.

Typically, $\eta = 0.01$.

But in Adagrad Optimizer the core idea is that **each weight has a different learning rate (η)**.

This modification has great importance, in the real-world dataset, some features are sparse and some are dense (most of the features will be non-zero), *so keeping the same value of learning rate for all the weights is not good for optimization.*

Adagrad [Adaptive Gradient]

* In SGD
  - ↳ Learning rate η is same for each weight ($\eta = 0.01$)

* But in adagrad, each weight/param-eter has different η
  - why?
  - ↓
  - Because features may be sparse or dense.
  - Sparse features tend to have different behaviour compared to dense features.

Adagrad

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}),$$

A derivative of loss function for given parameters at a given time t.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

Update parameters for given input i and at time/iteration t

η is a learning rate which is modified for given parameter **θ(i)** at a given time based on previous gradients calculated for given parameter **θ(i).**

Same for all weights

$W_{new} = W_{old} - \eta g_t$ → SGD

$W_{new} = W_{old} - \eta'_t g_t$ → Adagrad.

↳ adaptive different for each weights, changes for each iteration

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \varepsilon}}$$

↳ Small +ve no

$$\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2$$

$$g_i = \left(\frac{\partial L}{\partial w}\right)_{old}$$

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_{t-1}}\right)^2 \quad \textit{summation of gradient square}$$

Disadvantage of Adagrad

α_{t-1} can become very large as $t \uparrow$.

as $t \uparrow$.

$$\alpha_{t-1} = \sum_{t=1}^{t-1} g_i^2$$

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \varepsilon}}$$

$t \uparrow \Rightarrow \alpha_{t-1} \uparrow\uparrow\uparrow \Rightarrow \eta'_t \downarrow\downarrow\downarrow$

then

$W_{new} = W_{old}$

Solution is Adadelta & RMS Prop.

$$\alpha_{t-1} = \sum_{t=1}^{t-1} g_i^2$$

↳ should not be made large.

Adagrad

Advantages:

1.Learning rate changes for each training parameter.
2.Don’t need to manually tune the learning rate.
3.Able to train on sparse data.

Disadvantages:

1.Computationally expensive as a need to calculate the second order derivative.
2.The learning rate is always decreasing results in slow training.

AdaDelta and RMSProp

- Both Adadelta and RMSprop (Root Mean Square Propagation) aim to overcome the limitations of fixed learning rates in traditional gradient descent methods.
- It makes them well-suited for training deep neural networks.
- The choice between these algorithms often depends on the specific characteristics of the problem and empirical performance on a given dataset.

AdaDelta & RMSprop

$$w_{new} = w_{old} - \eta'_t g_t \rightarrow \text{Adadelta}$$

$$\eta'_t = \frac{\eta}{\sqrt{eda_{t-1} + \varepsilon}}$$

$$eda_{t-1} = \gamma\, eda_{t-2} + (1-\gamma) g^2_{t-1}$$

Take $\gamma = 0.95$

$$eda_{t-1} = 0.95\, eda_{t-2} + 0.05\, g^2_{t-1}$$

$$eda_{t-1} = 0.05 g^2_{t-1} + 0.95\, eda_{t-2}$$

$$= 0.05 g^2_{t-1} + 0.95 \left[0.95\, eda_{t-3} + 0.05 g^2_{t-2} \right]$$

$$= \underbrace{0.05 g^2_{t-1}}_{\text{most recent gets 5\% of weight}} + 0.95 \times 0.05 g^2_{t-2} + (0.95)^2 eda_{t-3}$$

$$\therefore \eta'_t = \frac{\eta}{\sqrt{eda_{t-1} + \varepsilon}}$$

↑ controls the growth of denominator term

∴ we dont get slow convergence here.

∴ Ada delta removes the problem of Adagrad of slow convergence.

RMSprop

- RProp algorithm does not work for mini-batches is because it violates the central idea behind stochastic gradient descent, when we have a small enough learning rate, it averages the gradients over successive mini-batches. To solve this issue, consider the weight, that gets the gradient 0.1 on nine mini-batches, and the gradient of -0.9 on tenths mini-batch, RMSProp did force those gradients to roughly cancel each other out, so that the stay approximately the same when computing.

- By using the sign of gradient from RProp algorithm, and the mini-batches efficiency, and averaging over mini-batches which allows combining gradients in the right way.

- RMSProp keep moving average of the squared gradients for each weight. And then we divide the gradient by square root the mean square.

The updated equation can be performed as:

$$E[g^2](t) = \beta E[g^2](t-1) + (1-\beta)(\frac{\partial c}{\partial w})^2$$

$$w_{ij}(t) = w_{ij}(t-1) - \frac{\eta}{\sqrt{E[g^2]}} \frac{\partial c}{\partial w_{ij}}$$

where $E[g]$ is the moving average of squared gradients, $\delta c/\delta w$ is gradient of the cost function with respect to the weight, η is the learning rate and β is moving average parameter (default value – 0.9, to make the sum of default gradient value 0.1 on nine mini-batches and -0.9 on tenths is approximate zero, and the default value η is 0.001 as per experience).

Adam

- Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks.
- First published in 2014, Adam was presented at a very prestigious conference for deep learning practitioners — ICLR 2015
- Adam can be looked at as a combination of **RMSprop and Stochastic Gradient Descent with momentum**.
- Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters.
- Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network

Adam

- The key idea behind Adam is to use a combination of momentum and adaptive learning rates to converge to the minimum of the cost function more efficiently.
- During training, it uses the first and second moments of the gradients to change the learning rate on the fly.
- The first moment is the mean of the gradients, and the second moment is the variance of the gradients.
- Adam maintains an exponentially decaying average of past and squared gradients and uses them to update the parameters in each iteration.
- This allows Adam to converge to the minimum of the cost function faster and more efficiently than traditional gradient descent methods.

Adam (Adaptive Moment Estimation)

Momentum + RMSProp is Adam.

$$w_{new} = w_{old} - \frac{\eta}{\sqrt{v_t + \varepsilon}} * m_t$$

η → momentum

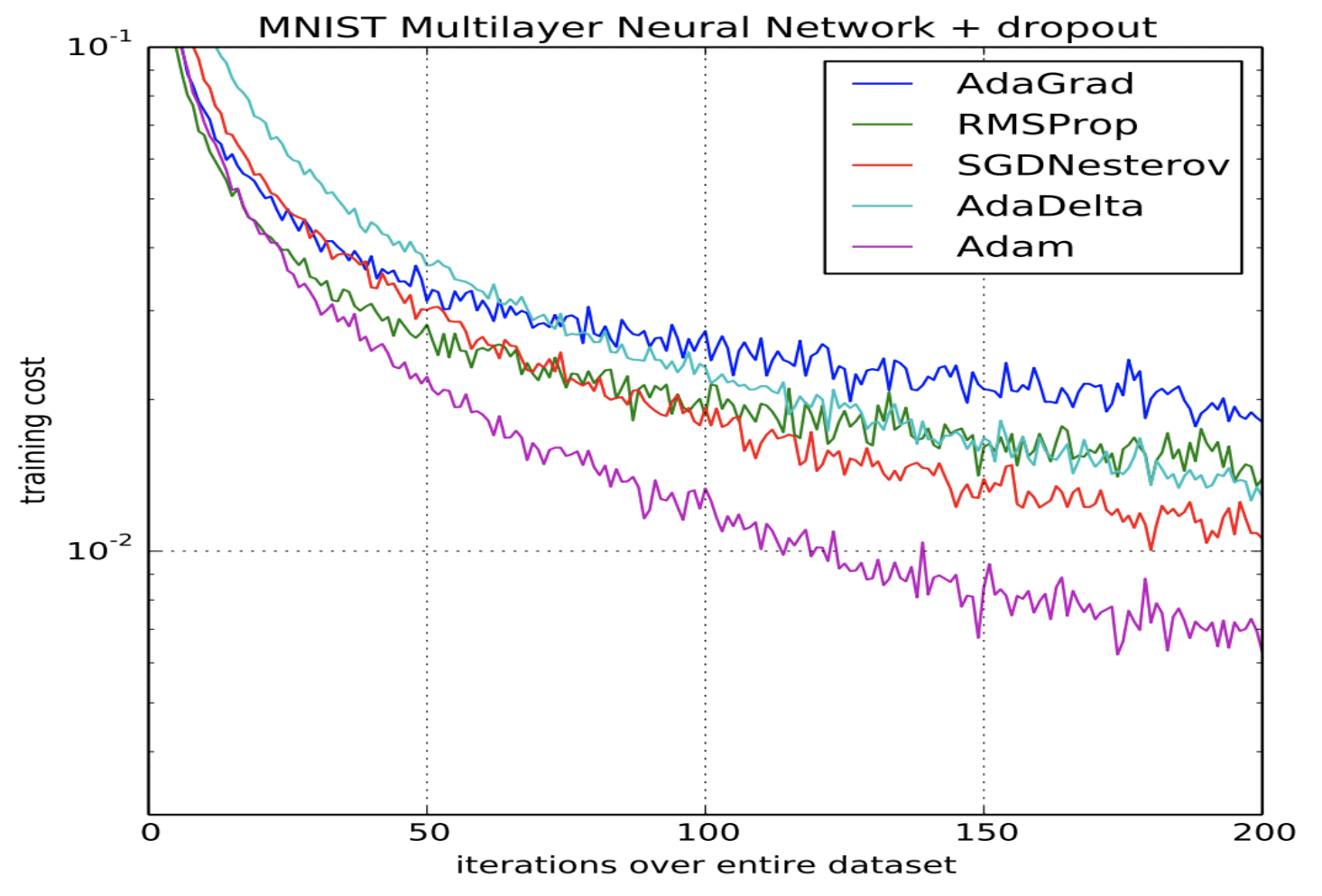
$\sqrt{v_t + \varepsilon}$ → RMSprop

$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w}$$

$$v_t = \beta v_{t-1} + (1-\beta)\left(\frac{\partial L}{\partial w}\right)^2$$

Adam (Adaptive Moment Estimation)

Momentum + RMSProp is Adam.



Adam

The weight updates are performed as:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

with

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

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

η is the step size/learning rate, around 1e-3 in the original paper. ϵ is a small number, typically 1e-8 or 1e-10, to prevent dividing by zero. β_1 and β_2 are forgetting parameters, with typical values 0.9 and 0.999, respectively.

Adam

- mt = Aggregate of gradients at time t [Current] (Initially, mt = 0)
- mt-1 = Aggregate of gradients at time t-1 [Previous]
- Wt = Weights at time t
- Wt+1 = Weights at time t+1
- αt = Learning rate at time t
- ∂L = Derivative of Loss Function
- ∂Wt = Derivative of weights at time t
- β = Moving average parameter (Constant, 0.9)

- Vt = Sum of the square of past gradients. [i.e sum(∂L/∂Wt-1)] (initially, Vt = 0)
- β = Moving average parameter (const, 0.9)
- ϵ = A small positive constant (10-8)

Adam

The main hyperparameters of Adam are the learning rate, beta1 (the exponential decay rate for the first moment estimate), and beta2 (the exponential decay rate for the second moment estimate).

These hyperparameters can be tuned for specific problems to achieve optimal results.

The key idea behind Adam is to use a combination of momentum and adaptive learning rates to converge to the minimum of the cost function more efficiently.

Adam

•**Adaptive learning rate** − Adam optimizer adaptively adjust the learning rate for each parameter which makes it suitable for problems with the sparse gradient or noisy gradient.

•**Fast Convergence** − Adam optimizer uses the momentum and the second moment of the gradients to speed up the convergence rate of the optimization process.

•**Efficient Memory Usage** − Adam optimizer maintains only two moving averages of the gradients, which makes it memory-efficient compared to other optimization algorithms that require the storage of a large number of past gradients.

Different Optimization Techniques

OPTIMIZER	DETAILS	ADVANTAGES	ISSUES
Batch Gradient Descent	Updates the parameters once the gradient of the entire dataset is calculated	Easy to compute, understand and implement	Can be very slow and requires large memory
Stochastic Gradient Descent	Instead of the entire dataset the calculation is done on few samples of data	Faster than BGD and takes less memory	Gradient results can be noisy and takes a lot of time to find minima
Mini-batch Gradient Descent	Splits whole dataset into subsets and parameters are updated after calculating the loss function of the subsets	Faster and more efficient than SGD	For too small learning rate, the process can be very slow and the updated gradients can be noisy
Momentum Based Gradient Descent	Reduces the noise of updated gradients and makes the process faster	Faster convergence and takes less memory	Computation of a new parameter at each update
Nesterov Accelerated Gradient	Moves toward the direction of past gradients, makes corrections and slowly approaches minima	Decreases the number of iterations and makes the process faster	Computation of a new parameter at each update
AdaGrad	Focuses on the learning rate and it can adjust according to the updates based on the sum of past gradients	Learning rate changes automatically with iterations	Massive decrease in learning rate can lead to slow convergence
AdaDelta	Adjusts learning rate based on the average of past squared gradients	Learning rate does not decrease massively	Computation cost can be high
Adam	Computation is based on both the average of past gradients and past squared gradients	Faster than others	Computation cost can be high

Saddle point problem

The term **“saddle point”** in the context of machine learning refers to a specific point in the optimization landscape of a cost function where the gradient is zero, but the point is neither a minimum nor a maximum.

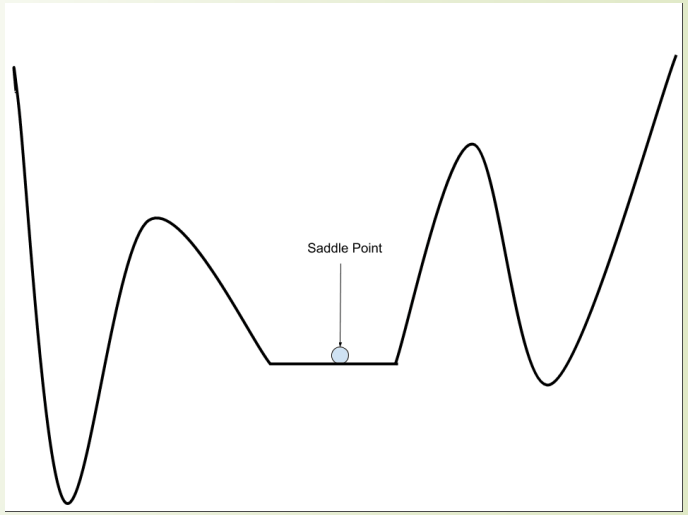
Why Saddle Points are a Challenge:

1. **Gradient Misleading:**

- The zero gradient at a saddle point can mislead optimization algorithms. While the algorithm thinks it has found a critical point, it may not be moving toward the actual minimum.

2. **Slow Convergence:**

- Optimization algorithms can converge very slowly around saddle points, as the small gradient values make it difficult for the algorithm to escape the region.

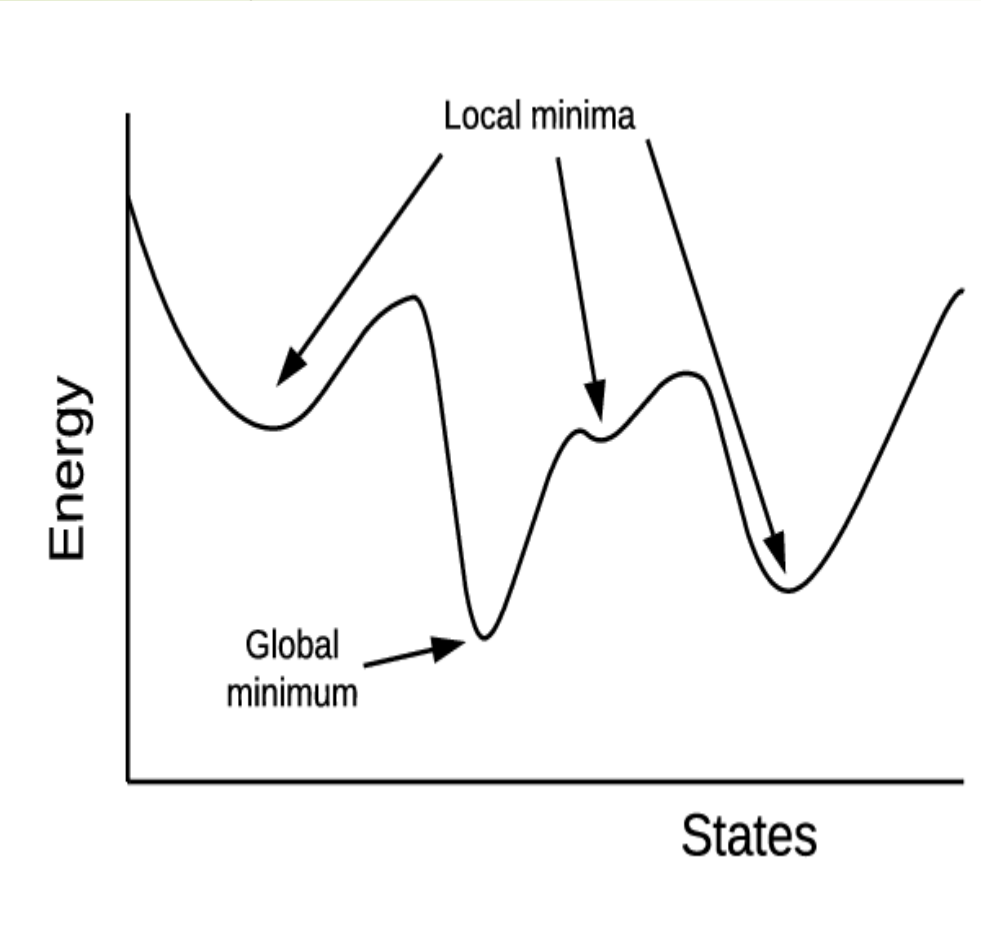


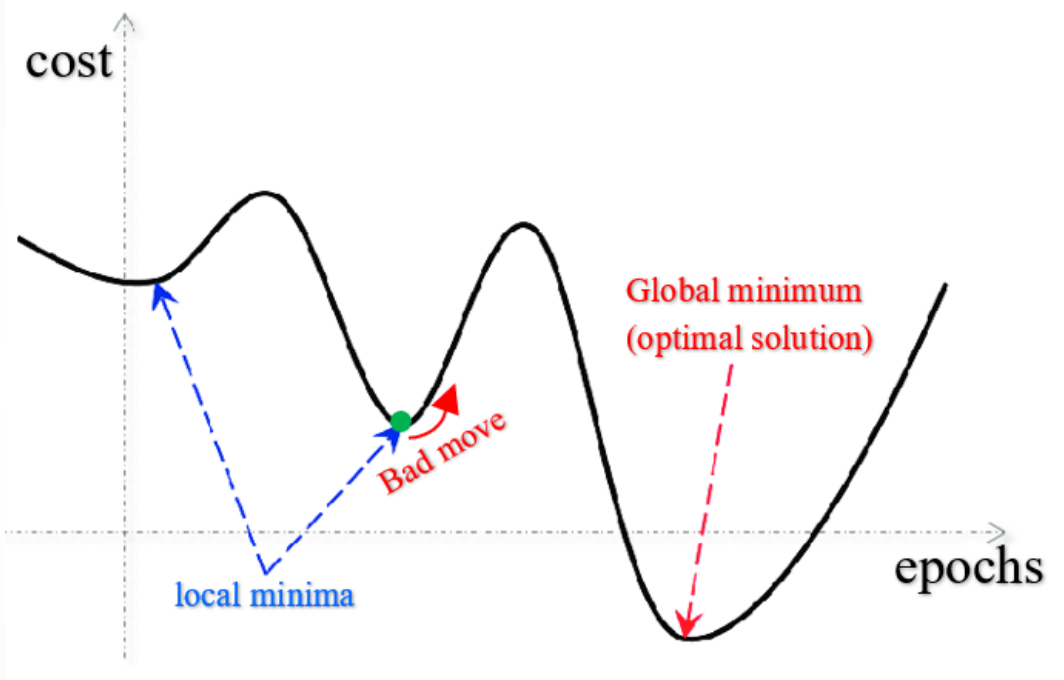
Best algorithm so far

- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- learning rate = 0.001 – 0.0001

ADAM and the above values of the hyperparameters are a very good starting point for any problem and virtually any type of neural network architecture I have ever worked with.

Local minima problem





Optimization Algorithms

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Optimization Algorithms

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Optimization Algorithms

Algorithm 8.5 The RMSProp algorithm

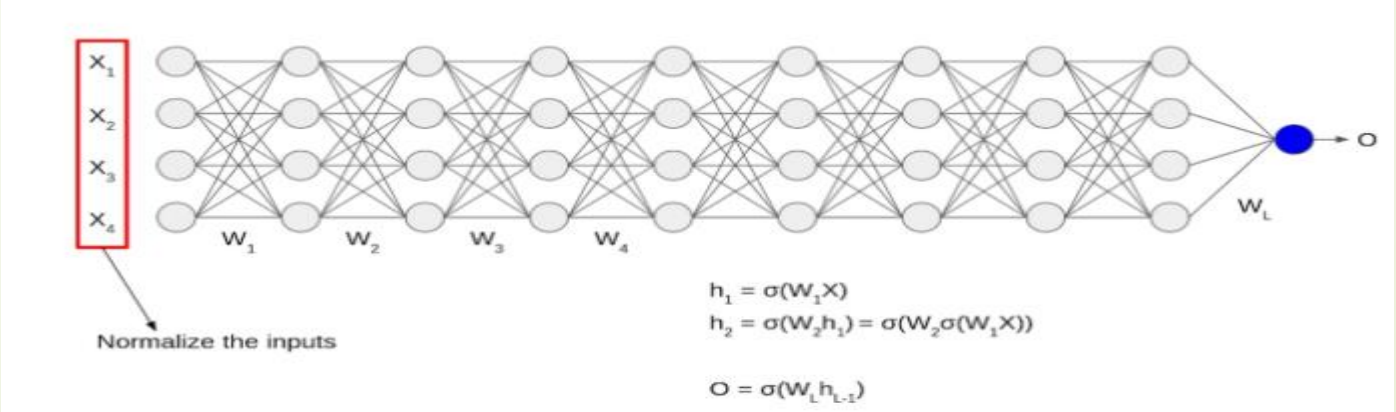
Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Algorithm 8.6: Basic Adam Optimizer

Require:
1: Initialized parameter θ_0 , step size η , batch size N_B
2: Exponential decay rates $\beta_1 , \beta_2 , \varepsilon$ dataset $\{(x_i , y_i)\}_{i=1}^{N}$
Initialize: $m_0 = 0 , \quad v_0 = 0$
3: **For** all $t = 1, \ldots \ldots , T$ ***do***
4: Draw random batch $\{(x_{ik} , y_{ik})\}_{k=1}^{NB}$ from dataset
5: $g_t \leftarrow \sum_{k=1}^{N} \nabla [\{(x_{ik} , y_{ik} , \theta_{t-1})\}$ // $f'(\theta_{t-1})$
6: $m_t \leftarrow \beta_1 . m_{t-1} + (1 - \beta_1) . g_t$ // moving Average
7: $v_t \leftarrow \beta_2 . m_{t-1} + (1 - \beta_2) . g_t^2$
8: $\hat{m_t} \leftarrow \frac{m_t}{1 - \beta_1^t}$, $\hat{v_t} \leftarrow \frac{v_t}{1 - \beta_2^t}$ // correction bias
9: $\theta_t \leftarrow \theta_{t-1} - \eta . \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \varepsilon}$
10: **end for**
11: **return** final parameter θ_T

Batch Normalization

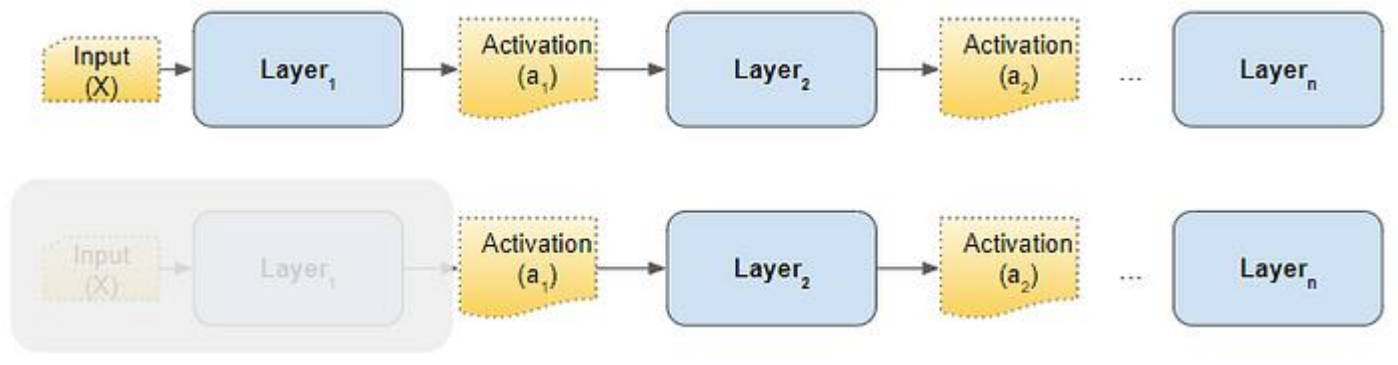
- ✓ Batch normalization (also known as batch norm) is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.
- ✓ Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.
- ✓ It was proposed by Sergey Ioffe and Christian Szegedy in 2015.



Although, our input X was normalized with time the output will no longer be on the same scale. As the data go through multiple layers of the neural network and L activation functions are applied, it leads to an internal co-variate shift in the data.

Batch Normalization

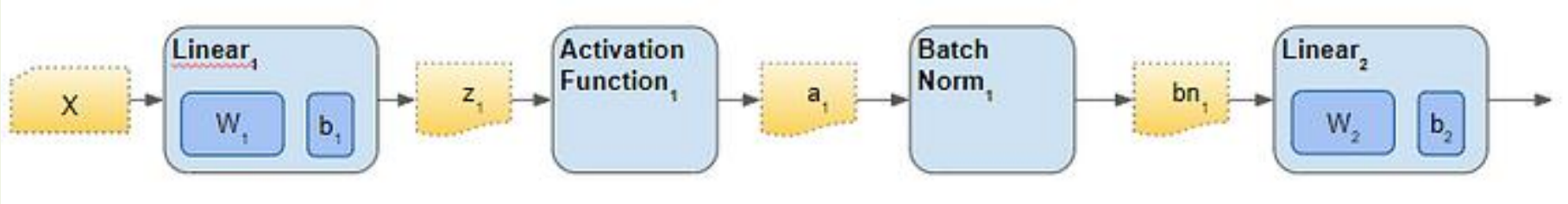
- Batch normalization is a supervised learning method for normalizing the interlayer outputs of a neural network.
- As a result, the next layer receives a “reset” of the output distribution from the preceding layer, allowing it to analyze the data more effectively.



The inputs of each hidden layer are the activations from the previous layer, and must also be normalized

Batch Normalization

Batch Norm is just another network layer that gets inserted between a hidden layer and the next hidden layer. Its job is to take the outputs from the first hidden layer and normalize them before passing them on as the input of the next hidden layer.

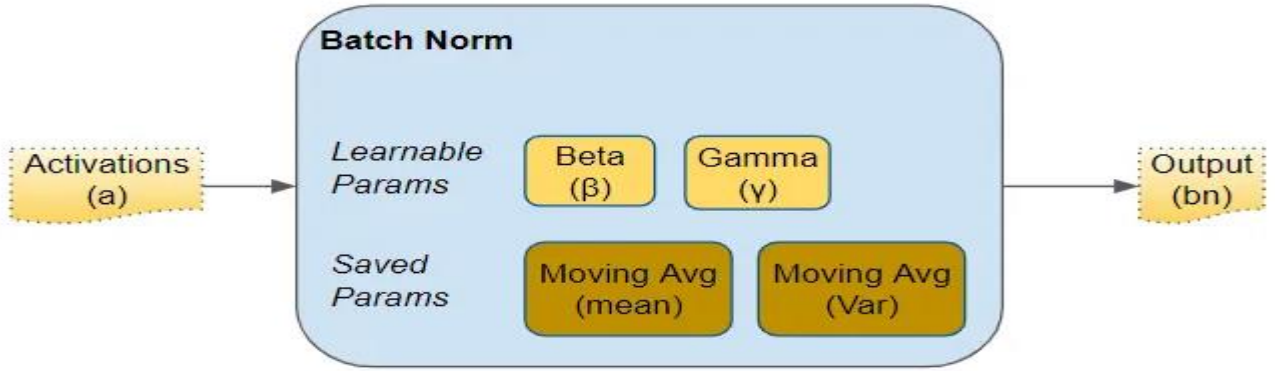


The Batch Norm layer normalizes activations from Layer 1 before they reach layer 2

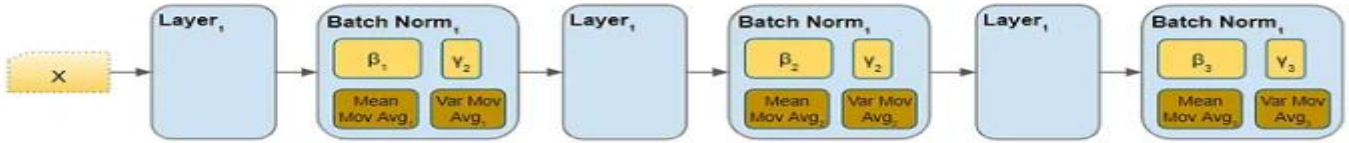
```
# we can think of this chunk as the input layer
model.add(Dense(64, input_dim=14, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))

# we can think of this chunk as the hidden layer
model.add(Dense(64, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
```

Batch Normalization

- Two learnable parameters called beta and gamma.
- Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the 'state' of the Batch Norm layer.



These parameters are per Batch Norm layer. So if we have, say, three hidden layers and three Batch Norm layers in the network, we would have three learnable beta and gamma parameters for the three layers. Similarly for the Moving Average parameters.



Batch Normalization

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

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

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

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

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

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

Batch Normalization- Advantages

- **Stabilize the training process:** Batch normalization can help to reduce the internal covariate shift that occurs during training, which can improve the stability of the training process and make it easier to optimize the model.
- **Improves generalization**: By normalizing the activations of a layer, batch normalization can help to reduce overfitting and improve the generalization ability of the model.
- **Reduces the need for careful initialization:** Batch normalization can help reduce the sensitivity of the model to the initial weights, making it easier to train the model.
- **Allows for higher learning rates:** Batch normalization can allow the use of higher learning rates that can speed up the training process.

Covariate Shift

- ✓ Covariate shift is a situation in which the distribution of the model's input features in production changes compared to what the model has seen during training and validation.
- ✓ Covariate shift is a change in the distribution of the model's inputs between training and production data
- ✓ Covariate shift is a common problem faced within the supervised type of machine learning methodology. It will occur when a model has been trained on a dataset with a distribution which is much different to new datasets. Because the distribution of input variables has shifted, the model may misclassify data points in a live environment.

Thank You