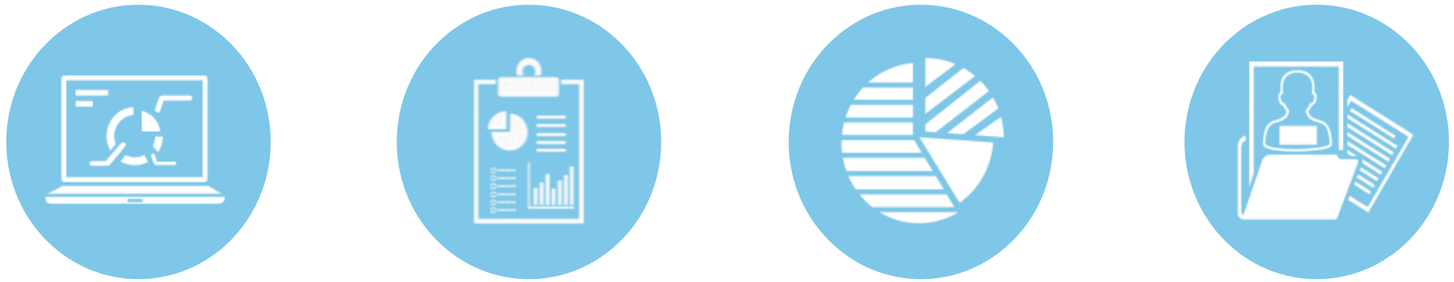
Deep Learning

Lesson 6— Training Deep Neural Nets



Learning Objectives

- Discuss solutions to speed up neural networks
- Explain regularization techniques to reduce overfitting

Training Deep Neural Nets

Topic 1—Developing Faster Neural Network

Problems in Deep Neural Networks

Assume a neural net of 10 layers, each containing hundreds of neurons connected by hundreds of thousands of connections.

The network is likely to cause the following problems:

It results in vanishing gradients (or exploding gradients) that make lower layers very hard to train.

It leads to slow training.

A model with millions of parameters would overfit the training set.

Vanishing/Exploding Gradient Problem

Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.

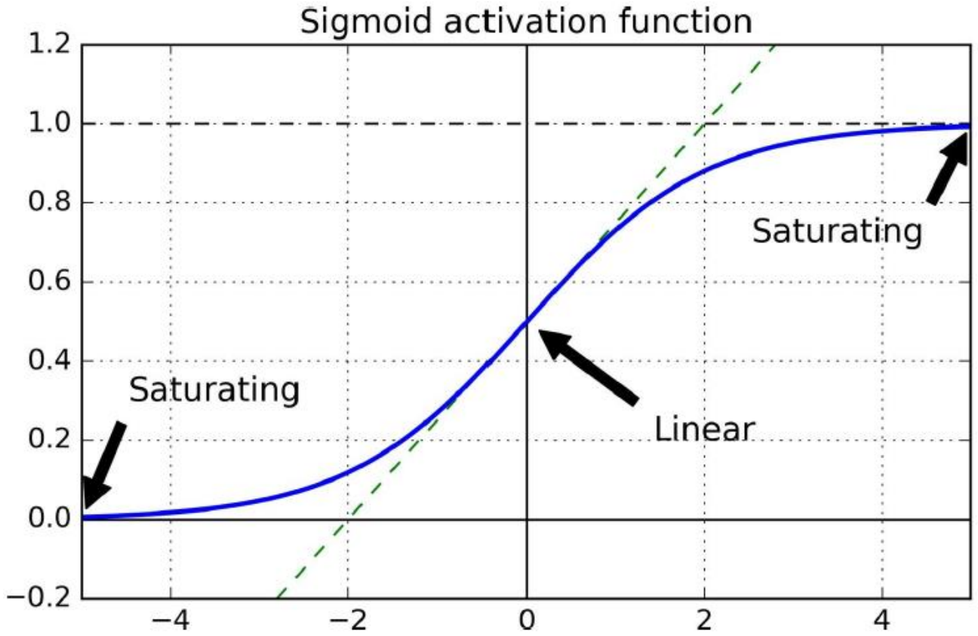
As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called vanishing gradients problem.

Weight adjustment is contingent on minimizing the cost function using gradient descent. So, if the gradients dilute (or explode), the gradient descent process fails to arrive at global minimum for loss. Without loss minimization, the model cannot be trained.

Vanishing/Exploding Gradient Problem

GRADIENTS IN A SIGMOID ACTIVATION FUNCTION

- The vanishing gradient problem is visible in the sigmoid graph, the most common activation function used.
- When the input is highly positive or negative, the response is close to 0 or 1 with little change. Derivative in these areas is very close to 0. So, during backpropagation (from right to left in a neural net), the gradient gets smaller and smaller toward lower layers, with little left of it finally.
- No gradient means no convergence.



Demo 1

Vanishing Gradient Problem

Objective: Demonstrate vanishing gradient in a sigmoid curve

Steps:

1. Analyze the tricky vanishing gradients problem (or the related exploding gradients problem) that affect deep neural networks and make lower layers very hard to train
2. Plot the sigmoid curve and review the vanishing gradients issue at its edges

Dataset used: None

Skills required: Vanishing Gradient problem and Sigmoid curve

How to Develop Faster Neural Nets?

The following techniques help solve vanishing / exploding gradient problem, as well as speed up the learning process:

- Applying a good initialization strategy for the connection weights (example: Xavier Initialization)
- Using a good activation function (example: ReLU activation)
- Using batch normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

XAVIER INITIALIZATION

- Researchers Glorot and Bengio have argued that for the signal to flow properly (forward and backward in a neural net) without vanishing/exploding gradients, the variance of the outputs of each layer needs to be equal to the variance of inputs. Also, the gradients should have equal variance before and after flowing through a layer in the reverse direction.

- If we assume that n_{inputs} and n_{outputs} refer to the number of input and output connections for the layer whose weights are being initialized, the weights must be initialized randomly as described below (when using logistic activation function):

$$\text{Normal distribution with mean 0 and standard deviation } \sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

$$\text{Or a uniform distribution between -r and +r, with } r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

This is called Xavier initialization or Glorot initialization.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

ReLU ACTIVATION FUNCTION

ReLU activation function is a non-saturating function (doesn’t saturate for positive values).

No saturation implies that the gradients do not dilute to zero, and one can use gradient descent to adjust weights in order to achieve convergence.

It provides faster convergence than Sigmoid function.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

LEAKY ReLU ACTIVATION FUNCTION

- ReLU suffers from dying ReLU problem: during training, the output of many neurons starts becoming 0 (caused by dead neurons). This happens even faster if one uses a large learning rate.
- A variant of ReLU, called a leaky ReLU, solves this problem. This is defined as:

$$\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$$

- The hyperparameter α defines how much the function "leaks." It is the slope of the function for $z < 0$ and is typically set to 0.01. This small slope ensures that leaky ReLUs never die.

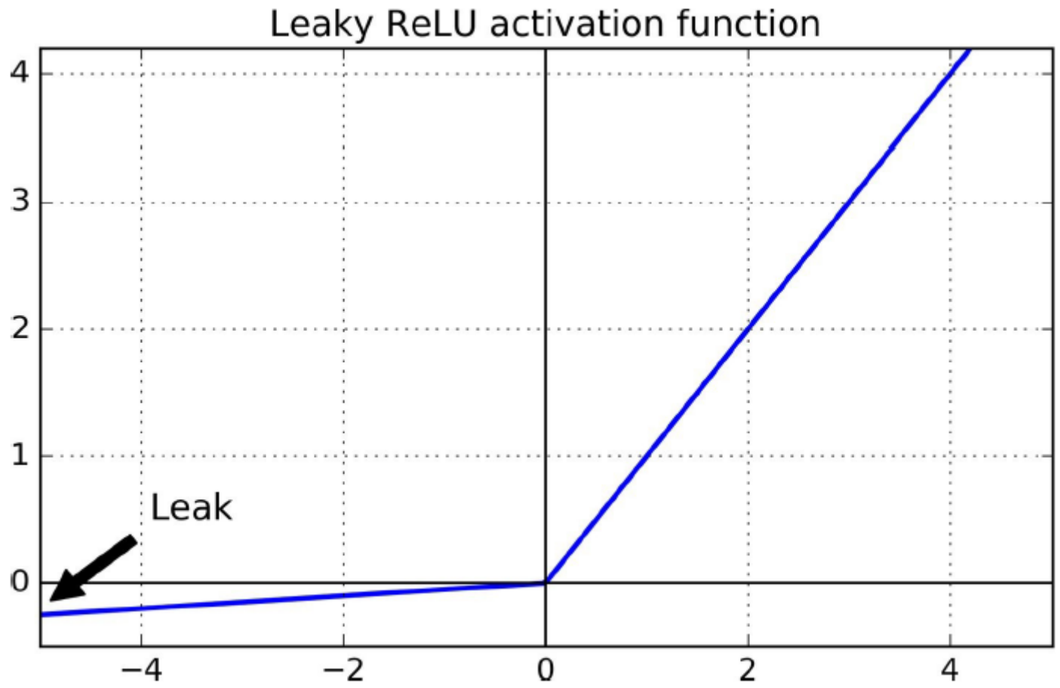


Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

ELU ACTIVATION FUNCTION

The exponential linear unit (ELU) was proposed in 2015 and outperformed all the ReLU variants in experiments.

- Training time was reduced
- Neural network performed better on the test set

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$

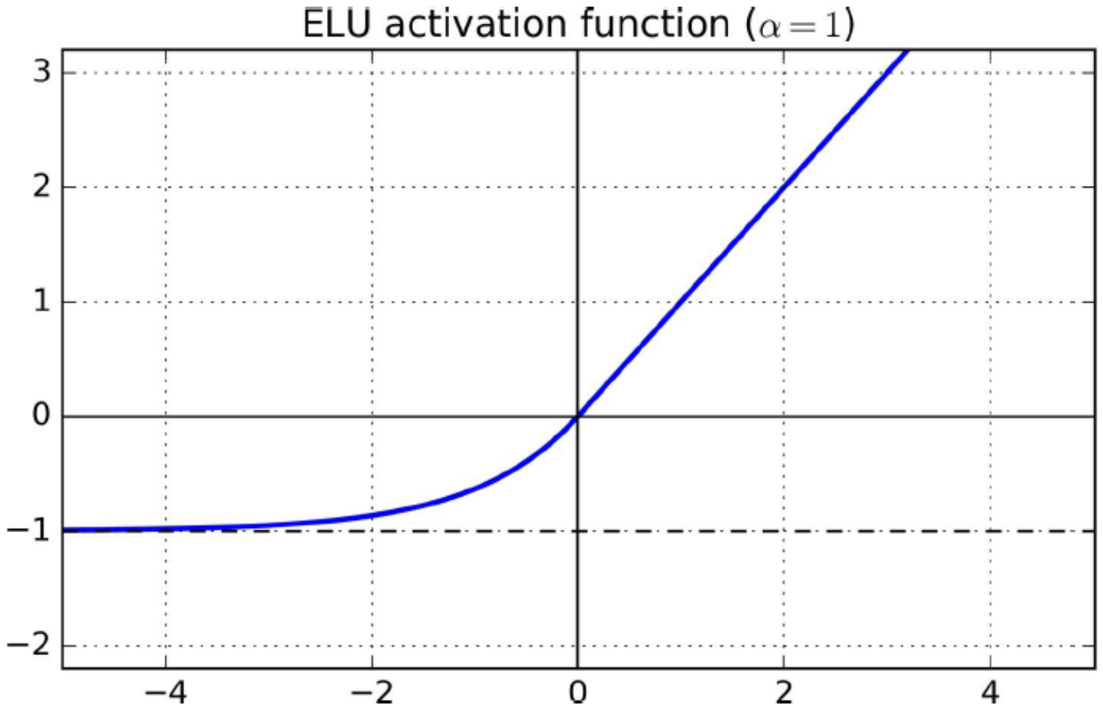


Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Solution for Faster Neural Network

Applying a good initialization strategy for the connection weights
Using a good activation function
Using Batch Normalization
Reusing parts of a pretrained network
Using a faster optimizer
Increasing the learning rate

ELU vs. ReLU

ELU is preferred over ReLU as:

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0. This helps alleviate the vanishing gradients problem.
- It has a nonzero gradient for $z < 0$, which avoids the dying units issue.
- The function is smooth everywhere, including around $z = 0$. This helps speed up Gradient Descent as it does not bounce as much left and right of $z = 0$.

Solution for Faster Neural Network

Applying a good initialization strategy for the connection weights

Using a good activation function

Using Batch Normalization

Reusing parts of a pretrained network

Using a faster optimizer

Increasing the learning rate

DEFINING LEAKY ReLU AND ELU IN TENSORFLOW

Leaky ReLU in Tensorflow:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu)
```

ELU in Tensorflow:

```
hidden1 = fully_connected(X, n_hidden1, activation_fn=tf.nn.elu)
```

In general: ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic"

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

BATCH NORMALIZATION

In a 2015 paper, Sergey Ioffe and Christian Szegedy proposed a technique called Batch Normalization (BN) to address the vanishing/exploding gradients problems.

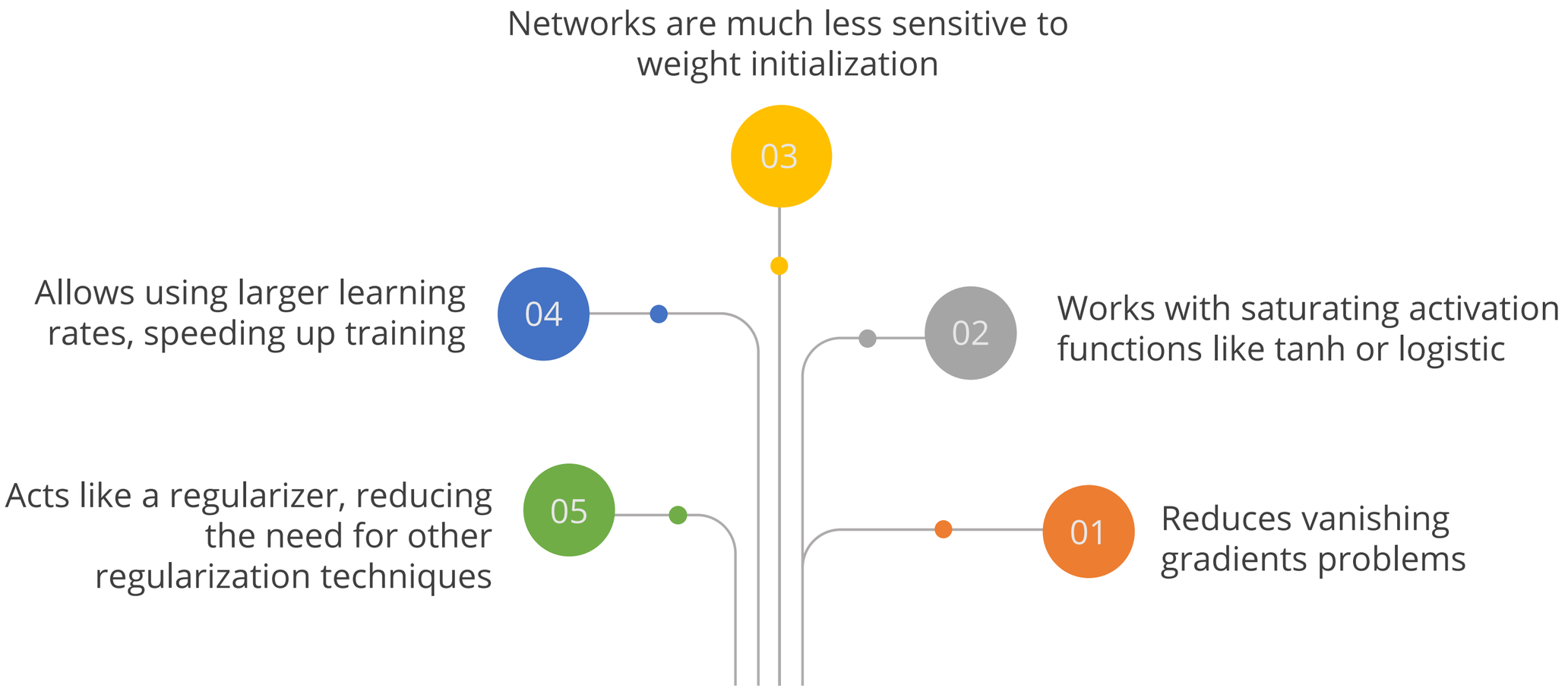
The technique consists of adding an operation in the model just before the activation function of each layer, zero-centering, and normalizing the inputs. It then involves scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting).

In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

BATCH NORMALIZATION: ADVANTAGES



Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

TRANSFER LEARNING

Pre-trained neural nets can be used in current problems by replacing part of the neural net with layers specific to the problem at hand. This is called Transfer Learning.

Solution for Faster Neural Network

Applying a good initialization strategy for the connection weights

Using a good activation function

Using Batch Normalization

Reusing parts of a pretrained network

Using a faster optimizer

Increasing the learning rate

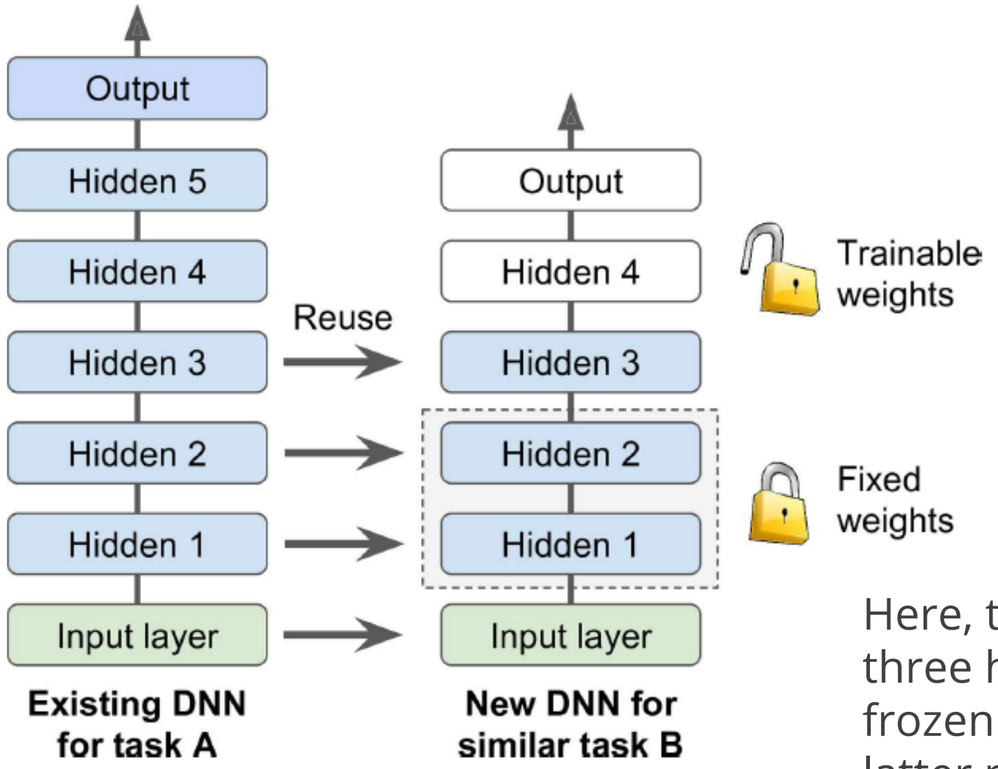
TRANSFER LEARNING: USING PRE-TRAINED LAYERS

- Transfer Learning involves using pre-trained layers from a prior trained neural network. This is a preferred practice for DNNs.
- It speeds up training and requires less training data.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

TRANSFER LEARNING: USING PRE-TRAINED LAYERS



Here, the weights of the first three hidden layers are said to be frozen (or fixed) when training the latter neural network.

If the input pictures of a new task don't have the same size as original task, add a preprocessing step to resize them to the size expected by the original model.

Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

REUSING A TENSORFLOW MODEL

To restore a pre-trained TensorFlow model and use it for new task, use the following command:

```
[...] # construct the original model

with tf.Session() as sess:
    saver.restore(sess, "./my_original_model.ckpt")
    [...] # Train it on your new task
```

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

REUSING A TENSORFLOW MODEL

To restore selected parts of the original model only (say, the first three layers), use the following command:

```
[...] # build new model with the same definition as before for hidden layers 1-3

init = tf.global_variables_initializer()

reuse_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[123]")
reuse_vars_dict = dict([(var.name, var.name) for var in reuse_vars])
original_saver = tf.Saver(reuse_vars_dict) # saver to restore the original model

new_saver = tf.Saver() # saver to save the new model

with tf.Session() as sess:
    sess.run(init)
    original_saver.restore("./my_original_model.ckpt") # restore layers 1 to 3
    [...] # train the new model
    new_saver.save("./my_new_model.ckpt") # save the whole model
```

The more similar the tasks are, the more layers you can reuse (starting with the lower layers). For very similar tasks, you can try keeping all the hidden layers and just replacing the output layer.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

MODEL ZOO

- The community has published many pre-trained models that can be reused. This is called a model zoo.
- TensorFlow has its own model zoo available at *https://github.com/tensorflow/models*. In particular, it contains most of the state-of-the-art image classification nets such as VGG, Inception, and ResNet (TensorFlow Slim package), including the code, the pretrained models, and tools to download popular image datasets.
- Another popular model zoo is Caffe's Model Zoo. There is a converter available from Caffe to TensorFlow.

Solution for Faster Neural Network

Applying a good initialization strategy for the connection weights

Using a good activation function

Using Batch Normalization

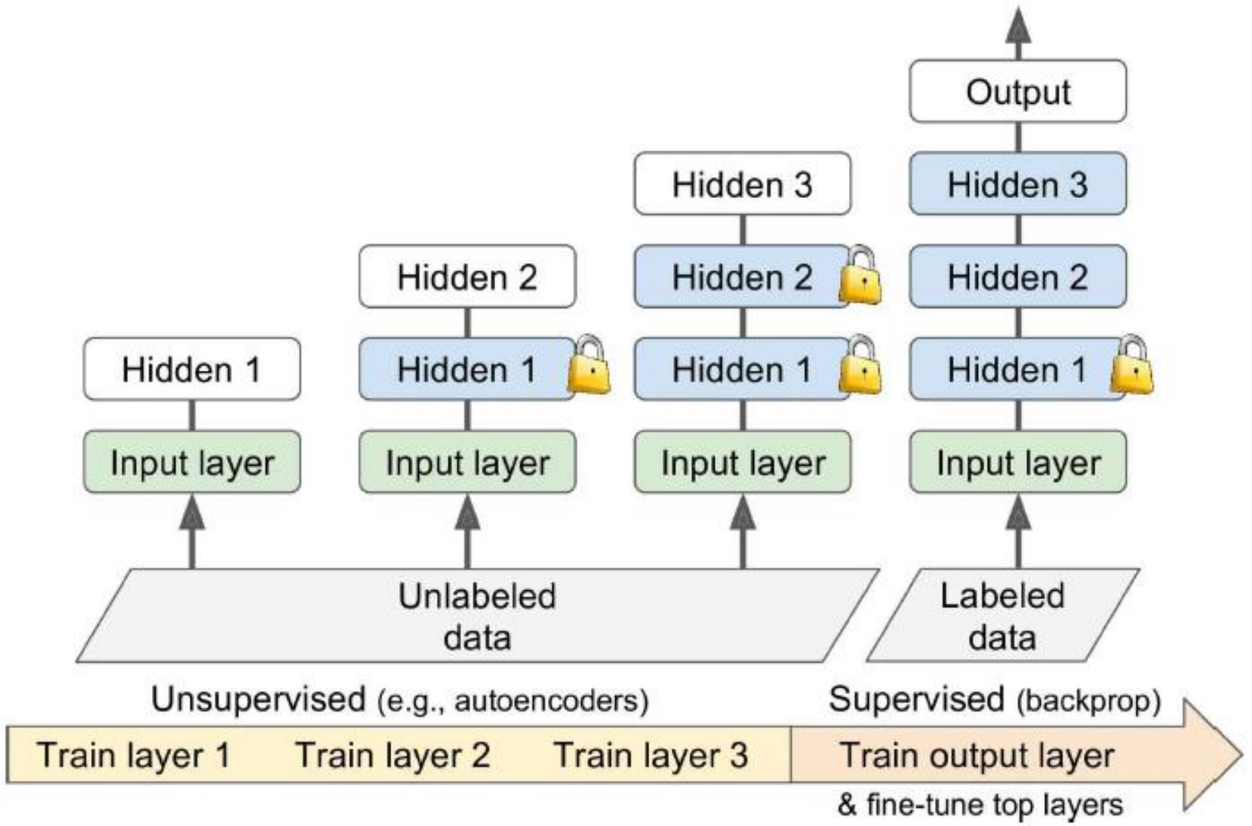
Reusing parts of a pretrained network

Using a faster optimizer

Increasing the learning rate

UNSUPERVISED PRETRANING

- If there is a shortage of labeled training data and plenty of unlabeled data, the lower layers can be pre-trained using a feature detector mechanism like AutoEncoders or Restricted Boltzmann Machines (RBMs).
- Each layer is trained on the output of the previously trained layers (all layers except the one being trained are frozen). Once all layers have been trained this way, one can fine-tune the network using supervised learning (i.e., with backpropagation).

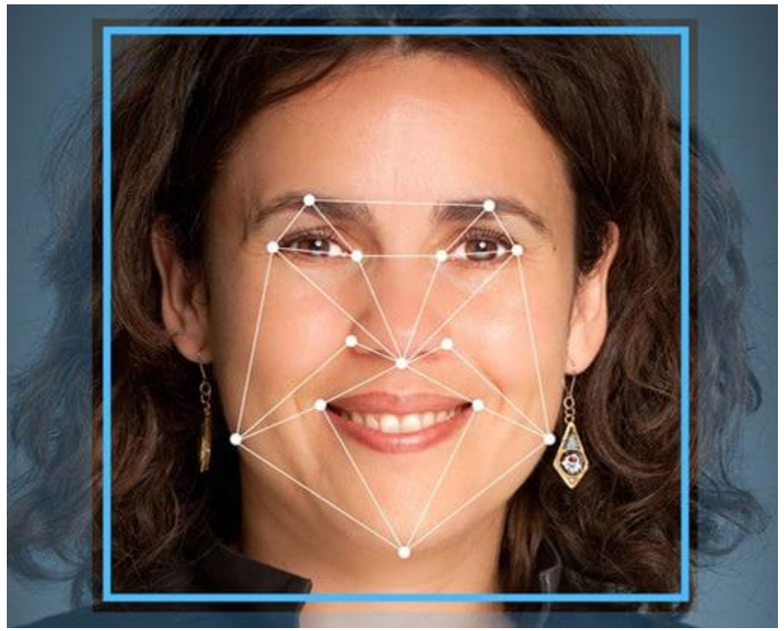


Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

PRE-TRANING ON AN AUXILARY TASK

If there is a shortage of training data of a certain type, pre-training on similar data can be done from other sources.



Example: For face detection of certain people, gathering hundreds of pictures per person may not be feasible.

In this case, pretraining can be done on a broader set of random people from the internet.

Such a network will learn good feature detectors for faces. Parts of this pretrained network can then be reused to train a model on a new set of face images.

https://www.spaceotechnologies.com/android-face-detection-example/

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

PRE-TRANING ON AN AUXILARY TASK

Training with unlabeled data

In order to pretrain layers for language modeling, one could follow this process:

- First, download millions of sentences from the internet
- Mark the sentences as "good"
- Corrupt some of the sentences (include wrong grammar or usage) and mark these corrupted ones as "bad"

Example: Consider a sentence “The dog sleeps.” Mark “The dog sleeps” as good. Mark “The dog they” as bad.

- A model trained to classify such good and bad sentences will learn about language, and its lower layers can be reused in different language tasks.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

FASTER OPTIMIZER

The following optimizers (apart from the usual GradientDescentOptimizer) help speed up training:

- Momentum optimization
- Nesterov Accelerated Gradient
- AdaGrad
- RMSProp
- Adam optimization

AdamOptimizer is widely used as it offers great performance.

Solution for Faster Neural Network

- Applying a good initialization strategy for the connection weights
- Using a good activation function
- Using Batch Normalization
- Reusing parts of a pretrained network
- Using a faster optimizer
- Increasing the learning rate

LEARNING RATE SCHEDULER

- High or low learning rates lead to problems.
- The best solution is to start with slightly higher learning rate and then reduce it gradually during training. There are some scheduling techniques to enable this.

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule.

Demo 2

Techniques to Improve Network Performance

Objective: Review solutions for increasing the network performance and solve vanishing/exploding gradient problem

Steps:

1. Demonstrate use of Xavier and He initialization.
2. Demonstrate use of Leaky ReLU and ELU activation.
3. Demonstrate use of pretrained layers stored in a checkpoint. Replace the 4th hidden layer from the pretrained model with a new hidden layer.
4. Plot TensorBoard output for pretrained layers.

Dataset used: MNIST dataset (available as part of TensorFlow install)

Skills required: Tuning options for deep neural nets e.g. vanishing/exploding gradients, good weight initialization, better activation functions like Leaky ReLU, and use of pretrained layers.

Training Deep Neural Nets

Topic 2—Regularization Techniques to Prevent Overfitting

Regularization

INTRODUCTION

- Regularization is any modification made to the learning algorithm that reduces its generalization error but not its training error.
- In addition to varying the set of functions or the set of features possible for training an algorithm to achieve optimal capacity, one can use other ways to achieve regularization.

Regularization

COST FUNCTION

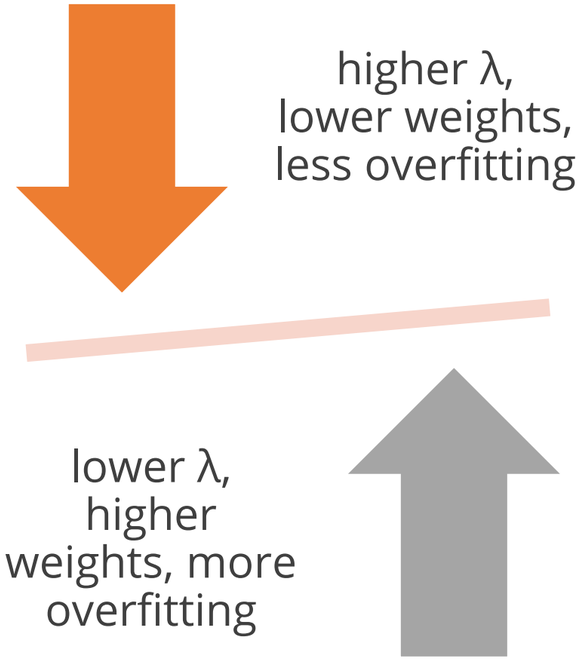
- One such method is weight decay, which is added to the Cost function.

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^{\top} \boldsymbol{w}.$$

- This approach not only minimizes the MSE (or mean-squared error) but also expresses the preference for the weights to have smaller squared L^2 norm (i.e., smaller weights).

- λ is a pre-set value. It influences the size of the weights allowed. A higher value of λ causes weights to be smaller in size (because minimizing the second term implies that if the pre-set value λ is large, smaller weights get learned, and vice versa)

- This works well as smaller weights tend to cause less overfitting (of course, too small weights may cause underfitting).



Regularization

REGULARIZER

- More generally, to regularize a model, a penalty is added to the Cost function. This is called a Regularizer: $\Omega(\mathbf{w})$
- Hence, the Cost function becomes $J(\mathbf{w}) = MSE_{train} + \lambda * \Omega(\mathbf{w})$
- In case of weight decay, this penalty is represented by $\Omega(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{w}$
- In essence, in the weight decay example, linear functions with smaller weights were preferred. This was done by adding an extra term to minimize the Cost function.

Avoiding Overfitting with Regularization

Regularization is a mechanism to introduce some kind of error in model training in order to prevent overfit to training data and create a more generalized model.

DNNs have tens of thousands of parameters, sometimes even millions. This can often lead to overfitting. Regularization techniques are used in preventing the overfitting issues:

- Early stopping
- $\ell 1$ and $\ell 2$ regularization
- Dropout

Avoiding Overfitting with Regularization

EARLY STOPPING

Early stopping
ℓ1 and ℓ2 regularization
Dropout

- In early stopping, the training is interrupted when performance on the validation set starts dropping.
- In TensorFlow, this can be implemented with some limit on the number of training steps.
- In practice, regularization is better when Early stopping is combined with other regularization techniques.

Avoiding Overfitting with Regularization

ℓ1 AND ℓ2 REGULARIZATION

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

- Regularization is used to constrain the neural network's weights (but typically not its biases).
- One way to do this in TensorFlow is to add appropriate regularization term to loss function. This works for one hidden layer.

```
[...] # construct the neural network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

- The code shows how you can add regularization loss (reg_losses) to the core loss function (base_loss).

Avoiding Overfitting with Regularization

ℓ1 AND ℓ2 REGULARIZATION

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

- For a higher number of layers, a more efficient way is to use the following command:

```
with arg_scope(
        [fully_connected],
        weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out")
```

- The function l1_regularizer(), l2_regularizer(), or l1_l2_regularizer() can be used.
- TensorFlow automatically adds the regularization nodes to a special collection containing all the regularization losses. Add regularization losses to the overall loss:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

Avoiding Overfitting with Regularization

DROPOUT

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

Dropout is the idea of dropping certain neurons from calculations altogether in subsequent passes of the training loops.

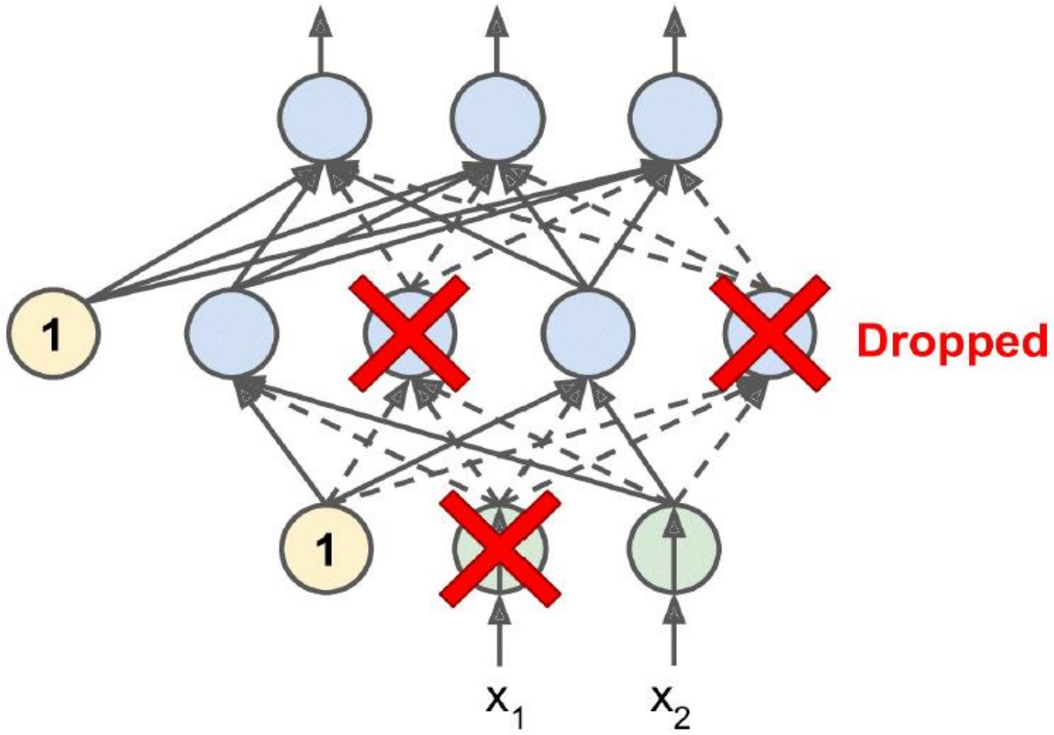
This has the effect of adding inaccuracy in the weight adjustments for various neuron layers, hence causing less overfitting for the provided training data.

Avoiding Overfitting with Regularization

DROPOUT

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

- Dropout is the most popular method for regularization.
- It gives high accuracy rate. A 95% accurate network can reach 96-97% accuracy rate, which is considered a big jump.
- In each training step, a few neurons from several of the layers (including input layer neurons or hidden layer neurons but not the output layer neurons) get dropped (not considered for subsequent processing). A neuron has a probability p of being dropped in a particular training step. This called dropout rate. Typically this value is set as p=50%.



Avoiding Overfitting with Regularization

WHY DROPOUT WORKS?

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

- In each pass of the training loop, a neuron can be included or excluded. Hence, surrounding neurons must become less sensitive to slight changes in inputs.
- With each pass of training loop, a new neural network is generated. In the end, we have a model trained on an ensemble of networks, which intuitively are able to generalize better.
- During training, neurons are trained with approximately half the signal, whereas during testing or production, neurons will receive almost double the signal. Hence, they are unlikely to perform well. So, one has to multiply connection weights with keep probability (1-*p*) after training OR during training and divide the output of neurons by the keep probability to mitigate this problem.

Avoiding Overfitting with Regularization

DROPOUT IN TENSORFLOW

The dropout function in TensorFlow randomly drops some items (setting them to 0) and divides the remaining items by the keep probability (during training only).

- Early stopping
- ℓ1 and ℓ2 regularization
- Dropout

```
from tensorflow.contrib.layers import dropout

[...]
is_training = tf.placeholder(tf.bool, shape=(), name='is_training')

keep_prob = 0.5
X_drop = dropout(X, keep_prob, is_training=is_training)

hidden1 = fully_connected(X_drop, n_hidden1, scope="hidden1")
hidden1_drop = dropout(hidden1, keep_prob, is_training=is_training)

hidden2 = fully_connected(hidden1_drop, n_hidden2, scope="hidden2")
hidden2_drop = dropout(hidden2, keep_prob, is_training=is_training)

logits = fully_connected(hidden2_drop, n_outputs, activation_fn=None,
                         scope="outputs")
```

Demo 3

Techniques to Prevent Overfitting Issues

Objective: Review various ways to regularize in order to reduce overfitting

Steps:

1. Demonstrate how l1 regularization is used to prevent overfitting.
2. Demonstrate Dropout. Use dropout_rate = 50%. Print accuracy on test dataset with dropout used during training.

Dataset used: MNIST dataset (available as part of TensorFlow install)

Skills required: The tuning options to reduce overfitting in deep neural nets e.g. regularization using l1/l2 and dropout mechanism.

Default DNN Configuration

The table below shows the effective set of tuning criteria for achieving efficient and fast neural networks:

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Key Takeaways

- Deep Neural Nets face a host of issues like vanishing/exploding gradients, slow learning, and overfitting.
- Xavier initialization for weight initialization help solve vanishing/exploding gradients.
- ReLU, Leaky ReLU and ELU are good activation functions for faster learning and non-saturating gradients.
- Batch normalization involves scaling and re-centering inputs to each layer. This helps in retaining gradients and faster learning.
- Transfer Learning involves using pretrained layers from a different network to speed up learning.
- AdamOptimizer is one of the best optimizers to use for fast learning.
- Regularization techniques like early stopping, l1 or l2 regularization, and dropout help prevent overfitting.

This concludes “Training Deep Neural Nets”

The next lesson is “Intro to Convolutional Neural Networks.”