



Chapter 11

Training Deep Neural Network



GitHub Page: [LINK](#)

Google Drive: [LINK](#)

- X_train and Y_train base code:

```
import sys
import tensorflow as tf
from tensorflow import keras
import numpy as np
import sklearn
import os
import matplotlib as mlp
import matplotlib.pyplot as plt

(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full = X_train_full / 255.0
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

pixel_means = X_train.mean(axis=0, keepdims=True)
pixel_stds = X_train.std(axis=0, keepdims=True)
X_train_scaled = (X_train - pixel_means) / pixel_stds
X_valid_scaled = (X_valid - pixel_means) / pixel_stds
X_test_scaled = (X_test - pixel_means) / pixel_stds
```

Training Deep Neural Network

- The previous chapter covered shallow nets but you may need to train high resolution images with much deeper DNN
- Training a deep DNN is not easy, here are some issues that might be encountered
 - vanishing or exploding gradient problem
 - not have enough data for a large network or it might be too costly to label
 - extremely slow training

- A model with millions of parameter risks overfitting the training set especially if there are not enough training instances or they are too noisy
- This chapter will go through some of the solutions to those issues

The Vanishing/Exploding Gradients Problems

- The network uses gradients to update each parameter with a gradient descent step
- gradients often gets smaller and smaller as the algorithm progresses down
 - And therefore the gradient descent update leaves the tower layers' connection weight virtually unchanged
 - And the training never converges to a good solution
 - This is called **vanishing gradient**
- The opposite may happen when gradient grows bigger and bigger until layers get insanely large weight updates and the algorithm diverges
 - This is called **exploding gradient**

Glorot and He Initialization

- We don't want the signal to die out nor we want it to explode and saturate
- It is not possible to guarantee both unless the layer has an equal number of input and neurons
- a good compromise that was proven to work well in practice:
 - the connection weights of each layer must be initialized randomly
 - where $fan_{avg} = (fan_{in} + fan_{out})$
 - This is called Xavier initialization or Glorot initialization after the papers of the first authors
- Some papers have provided similar strategies for different activation functions
 - differ only by scale of the variance and whether they use fan_{avg}



Table shown at page 334

- By default Keras use Glorot initialization with a uniform distribution
- can change this to `kernel_initializer= "he_uniform"` or `"he_normal"`

Non-saturating Activation Functions

- The paper shows that the reason with unstable gradient is because of the poor choice of activation functions
- ReLU is not perfect and it suffers from a problem known as dying
 - In training some neurons die and effectively output 0
- A neuron die when its weights is tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set
 - When this happens it just keeps outputting 0
- To solve this issue:
 - Use leaky ReLU
 - The leak insures that the function never dies
 - There's different kinds of leaky ReLU
 - Randomized leaky ReLU (RReLU)
 - Parametric leaky ReLU (PReLU)
 - exponential linear unit (ELU) outperforms all ReLU based activation functions
- ELU is similar to ReLU with a few differences:
 - Takes on negative values when $z < 0$ which allows it to have a output closer to 0 and helps with vanishing gradient issue
 - Nonzero gradient for $z < 0$
 - if $\alpha = 1$ then the function is smooth everywhere including around $z = 0$
 - Helps with speed up gradient descent since it doesn't bounce back as much
- ELU is slower to compute than ReLU
- 2017 introduced Scaled ELU, it's a scaled variant of ELU
- If all hidden layers are composed of SELU with stacks of Dense layers then the network will self-normalize

- The output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training
- Which solves vanishing/exploding gradients
- Therefore SELU usually outperform other activation function in neural net especially deep ones
- Few condition for self normalization to happen:
 - input features must be standardized
 - every hidden layer's weight must be initialized with LeCun normal initialization
 - `kernel_initializer="lecun_normal"`
 - network architecture must be sequential
 - If used for recurrent networks or skip connections, self normalization is not guaranteed
 - All layers must be dense but some research suggest SELU improves performance in convolutional neural nets as well
- Using leaky ReLU

```
model = keras.models.Sequential([
    #[...]
    keras.layers.Dense(10, kernel_initializer="he_normal")
    keras.layers.LeakyReLU(alpha=0.2)
    #[...]
])
```

- Using PReLU is similar

```
model = keras.models.Sequential([
    #[...]
    keras.layers.Dense(10, kernel_initializer="he_normal")
    keras.layers.PReLU(alpha=0.2)
    #[...]
])
```

- Using SELU is different

```
layer = keras.layers.Dense(10, activation="selu", kernel_initializer="lecun_normal")
```

Batch Normalization

- Although some variation of ReLU (ELU) prevent exploding/vanishing gradient at the start of the training. It doesn't guarantee it won't come back
- a paper from 2015 introduced a technique called batch normalization that addresses these issues
- The technique is to adding an operation in the model just before or after the activation function in each hidden layer
- The operation lets model learn the optimal scale and mean of each layer's inputs
- If BN is added to the first layer then it doesn't require manually standardize training, BN automates it
- The algorithm estimates each input's mean and standard deviation
- It does so by evaluating the mean and standard deviation of the input over current mini batches



Algorithm on page 339

- link to YouTube video that explains BN: [Link](#)

Implementing Batch Normalization with Keras

- In Keras It's simple and intuitive to add BN
- Simply add "BatchNormalization" layer before or after each hidden layer's activation function

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

- Not likely to have too much impact due to it only having two layers
- Displaying summary

```

model.summary()
'''
Model: "sequential_4"

```

Layer (type)	Output Shape	Param #
flatten_4 (Flatten)	(None, 784)	0
batch_normalization (Batch Normalization)	(None, 784)	3136
dense_212 (Dense)	(None, 300)	235500
batch_normalization_1 (Batch Normalization)	(None, 300)	1200
dense_213 (Dense)	(None, 100)	30100
batch_normalization_2 (Batch Normalization)	(None, 100)	400
dense_214 (Dense)	(None, 10)	1010

```

Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
'''

```

- To look at parameter in the first BN layer

```
[(var.name, var.trainable) for var in bn1.variables]
```

- When create BN layer in Keras, it also creates 2 operations that will be called by Keras during each iteration
 - It will update the moving averages

```
bn1.updates
```

- Batch normalization layers includes one offset bias parameter, to remove it:
 - pass in use_bias=False

```

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(100, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),

```

```
keras.layers.Dense(10, activation="softmax")
])
```

- There are hyperparameters to tweak but usually the default would be fine
 - teak momentum
 - This hyperparameter is used by the BatchNormalization layer when it updates the exponential moving averages
 - Typically close to 1 (0.9, 0.99, 0.999)
 - Tweak axis
 - Determines which axis to normalize, default is -1
- The "call()" method is the one that performs the computations
 - fit() sets default training argument as 1 but call() sets it as None

Gradient Clipping

- Clipping the gradient so they never exceed a threshold
- In Keras, implementing gradient clipping is as simple as setting the "clipvalue" or "clipnorm" argument while creating optimizer

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- Clip every value between -1 to 1, this is a hyperparameter to tune

Reusing Pretrained Layers

- It's not good to start large DNN from scratch so it's good idea to find existing neural network that does similar task
- Using pretrained layers from other network and transferring it to yours is called **transfer learning**
- speeds up training and requires less training data
- During transfer learning most likely the lower layers are more useful
- While using layers from other network freezing the network is a good idea since by doing that it stops the layer from further training

- can try different combination of freezing and training layers to get the most optimal results

Transfer Learning with Keras

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ca567739-da6d-496d-84f8-02ed00291245/transfer_learning.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ab2d05ca-be6a-4bba-bf3f-55e59a603103/transfer_learning.ipynb

Unsupervised Pretraining

- If there is lots of labeled data, then it's still possible to perform unsupervised learning
- you can train an autoencoder or a generative adversarial network
- A method to doing this is called: **greedy layer-wise pretraining**
- first train unsupervised model with a single layer
- Then add more hidden layers onto it with unsupervised algorithm



Graph on page 350

Pretraining on Auxiliary Task

- If there isn't much labeled data then the last option is to train a first neural network on an auxiliary task which could be easily obtain or generate labeled training data
- An example would be a face detection system, instead of taking lots of pictures of people, it's an option to create a network from images online and then use the network's layer to create the new network

Faster Optimizers

- Training neural network is slow, to make the training faster we've so far seen:
 - good initialization strategy for connection weights
 - good activation function
 - batch normalization
 - reusing parts of a pretrained network
- Good optimizer also increases training speed
- Popular optimizers are: Nesterov accelerated gradient, momentum optimization, AdaGrad, RMSProp, Adam and Nadam optimization

Momentum Optimization

- It's like rolling a ball down the mountain, starts slow and then increases by speed
- the gradient is used for acceleration not for speed
- Momentum algorithm:

$$m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta + m$$

- Implementing momentum in Keras is simple, just use SGD optimizer and set its momentum hyperparameter

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Accelerated Gradient

- It's a small variant to the momentum optimization
- Nesterov Accelerated Gradient algorithm

$$m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m)$$

$$\theta \leftarrow \theta + m$$

- Slightly more accurate compared to momentum
- To implement this in Keras, just enable nesterov:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad

- AdaGrad algorithm achieves correction by scaling down the gradient vector along the steepest dimensions
- AdaGrad algorithm

$$s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon}$$

- Usually stops too early when training neural networks therefore it's not a good idea to use it
- Although Keras has an Adagrad optimizer

RMSProp

- RMSProp algorithm fixes the risk of slowing down a bit too fast as AdaGrad by accumulating only the gradients from the most recent iterations
- RMSProp algorithm:

$$s \leftarrow \beta s + (1 - \beta_1) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon}$$

- decay rate is typically set to 0.9, this is a hyperparameter
- lr is learning rate
- To implement this in Keras:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam and Nadam Optimization

- Adam stands for adaptive moment estimation combines momentum optimization and RMSProp
- It keeps track of an exponentially decaying average of past gradients and it keeps track of an exponentially decaying average of the past squared

gradients

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\hat{m} \leftarrow \frac{m}{1 - \theta_1^t}$$

$$\hat{s} \leftarrow \frac{s}{1 - \theta_2^t}$$

$$\theta \leftarrow \theta - \eta \hat{m} \oslash \sqrt{\hat{s} + \varepsilon}$$

- β_1 is momentum decay hyperparameter
- β_2 is the scaling decay hyperparameter
- To use Adam in Keras

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

AdaMax

- It's a variation of Adam
- AdaMax replaces l_2 with l_{∞} norm
- Replaces step 2 with $s \leftarrow \max(\beta_2 s, \nabla_{\theta} J(\theta))$
- Drops step 4 and 5 and scales down the gradient updates by a factor of s which is just the max of the time-decayed gradient
- This could make AdaMax more stable than Adam in practice, but really depends

Nadam

- Nadam optimization is Adam optimization plus Nesterov trick
- converge slightly faster than Adam

Learning Rate Scheduling

- Finding good learning rate is important
- If it's too high then it diverges or it might be suboptimal
- If it's too low then the training is going to be too slow

- If the learning rate is just right then it saves compute power and time
- There's better ways than constant learning rate
 - Start high and reduce it once training stops making fast progress
- These strategies are called learning schedules:

Power scheduling

- The learning rate drops at each step
- The algorithm is:

$$\eta(t) = \eta_0 / (1 + t/s)^c$$

- Initial learning rate is η_0
- power c is usually set to 1
- s is steps
- The 3 things listed above are hyperparameters
- The learning rate drops quickly as time increases

Exponential scheduling

- Algorithm:

$$\eta(t) = \eta_0 \times 0.1^{t/s}$$

- The learning rate will gradually drop by factor of 10 every s steps

Piecewise constant scheduling

- Using a constant for a certain amount of epochs
- For example:
 - $\eta_0 = 0.1$ for 5 epochs
 - $\eta_0 = 0.0001$ for 50 epochs

Performance Scheduling

- measures validation error every N steps and reduce learning rate by a factor of λ when the error starts dropping

1-cycle Scheduling

- Increase linearly at the start of training
- Starts decreasing linearly after half of training
- Finishing last few epochs by dropping down learning rate by several orders of magnitude (linear)
- Max learning rate η_1 is chosen using the same approach used to find optimal learning rate
- initial learning rate η_0 is chosen to be roughly 10 times lower
- With this scheduler using CIFAR10 dataset, it managed to reach 91.9% accuracy in 100 epochs instead of 90.3% accuracy in 800 epochs

Implementing in Keras

- Implementing power scheduling

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ead13528-b41b-43aa-b32a-a003f06f6808/power_scheduling.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/224a01d0-d91-4324-8459-77d69c8c9616/power_scheduling.py

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

- Exponential scheduling

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/64765469-358d-432d-82dd-2341cd6ecc0f/exponential_scheduling.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1a77f722-4b2b-4cd9-805c-10b064ea9bf9/exponential_scheduling.ipynb

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1 ** (epoch/20)
```

- To hard code η_0 and s

```
def exponential_decay(lr0, s):
    def expontial_decay_fn(epoch):
        return lr0 * 0.1 ** (epoch/s)
    return expontial_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

- create learning rate scheduler

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

- If scheduler relies on epochs, then saving and continuing model would not be so simple since number of epochs are not saved
- A solution is to manually set the fit() method's initial_epoch argument so it starts at the right value
- For piece wise scheduling

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

- For performance scheduling callback

```
lr_scheduler = keras.callbacks.ReduceLRonPlateau(factor=0.5, patience=5)
```

- The code above will multiply the learning rate by 0.5 whenever best validation loss does not improve for 5 epochs
- Keras offers exponential decay schedulers

```
s = 20*len(X_train) / 32
learning_rate = keras.optimizers.schedulers.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

Avoiding Overfitting Through Regularization

- a typical network tens of thousands of parameters
- This give it an incredible amount of freedom
- This flexibility also causes overfitting the training set

ℓ_1 and ℓ_2 Regularization

- ℓ_1 regularization for sparse model
- ℓ_2 regularization to constrain a neural network's connection weights

```
layer = keras.layers.Dense(100, activation='elu', kernel_initializer='he_normal',
    kernel_regularizer=keras.regularizer.l2(0.01))
```

- `l2()` function returns a regularizer that will be called at each step during training to compute the regularization loss
- Could also use `l1()` for ℓ_1 regularization
- Since this will be repeating multiple times, refactoring the code to use loops will be more efficient

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense, activation='elu',
    kernel_initializer='he_normal', kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation='softmax', kernel_initializer='glorot_uniform')
])
```

Dropout

- One of the most popular techniques in deep neural networks
- gets around 1-2% accuracy boost simply by adding dropouts

- A percentage of layers excluding output layers will be temporarily "Dropped out" during training
- The hyperparameter dropout rate is typically between 10-50 %
- closer to 40-50% for convolutional network
- The reason dropouts work is because the neurons will not rely on any other neurons because there's a chance of them not appearing in the next epoch
- To implement dropout in Keras

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation='elu', kernel_initializer='he_normal'),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation='elu', kernel_initializer='he_normal'),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation='softmax')
])
```

- To observe if the model is overfitting, you can increase the dropout rate
- Try to decrease the dropout rate if the model is underfitting
- Increasing dropout rates for large layers and decreasing for small layers may also help
- Slows down convergence but generally worth it

Monte Carlo (MC) Dropout

- MC Dropout can increase performance of any trained dropout model without having to retrain it or modify it at all
- To implement this:

```
y_probas = np.stack([model(X_test_sacled, training=True) for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

- MCDropout class

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```


Max-Norm Regularization



- Another regularization technique that is popular for neural networks is Max-Norm Regularization
- For each neuron it constrains the weight w of the incoming connection such that $\|w\|_2 \leq r$
- r is hyperparameter and $\|\cdot\|_2$ is the ℓ_2 norm
- Does not add a regularization loss term to the over all loss function
- Implemented by computing $\|w\|_2$ after training and rescaling w if needed
- To implement this in Keras:

```
keras.layers.Dense(100, activation='elu', kernel_initializer='he_normal',  
kernel_constraint=keras.constraints.max_norm(1.))
```

- Max-Norm axis has a default of 0

Summary

DNN configuration

 Hyperparameter	 Default Value
<u>Kernel initializer</u>	He initialization
<u>Activation Function</u>	ELU
<u>Normalization</u>	None if shallow; Batch Norm if deep
<u>Regularization</u>	Early stopping
<u>Optimizer</u>	Momentum optimization (or RMSProp or Nadam)
<u>Learning rate schedule</u>	1cycle

Exercises

1. No the weights cannot be at the same value it must be randomly initialized.
2. Yes it is, could even remove bias completely
3. Output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training. SELU solves vanishing/exploding gradients and SELU usually outperform ReLU and other activation function in deep neural networks

4.

SELU: It is a good default

Leaky ReLU: If you want the network to perform as well as possible

ReLU: It is one of the most simple activation functions, and it could be useful when you need 0 as output

tanh: Useful in output layer when you want a number from -1 to 1, rarely used in hidden layers

logistic: Useful in output layer when you need to estimate probability, rarely used in hidden layers

softmax: Used in output layer to output probability of mutually exclusive classes

5. If you set momentum hyperparameter too close to 1 it will pick up speed really fast and then it's going to come back down after half of training and it's likely to overshoot therefore it's going to take much longer to converge

6. train model normally then zero out small weights, or applying ℓ_1 normalization

7. Yes, dropout slows down training. It has no effect on inference. MC Dropout is a way to apply dropout without retraining the model, although it's not required to retrain the model, it will slow down inference

8. Files:

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e197be33-297e-4666-9b7e-cdf879db6e66/cifar10.ipynb>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0592cd00-8c20-48e6-bfc8-6def270c472d/cifar10.py>