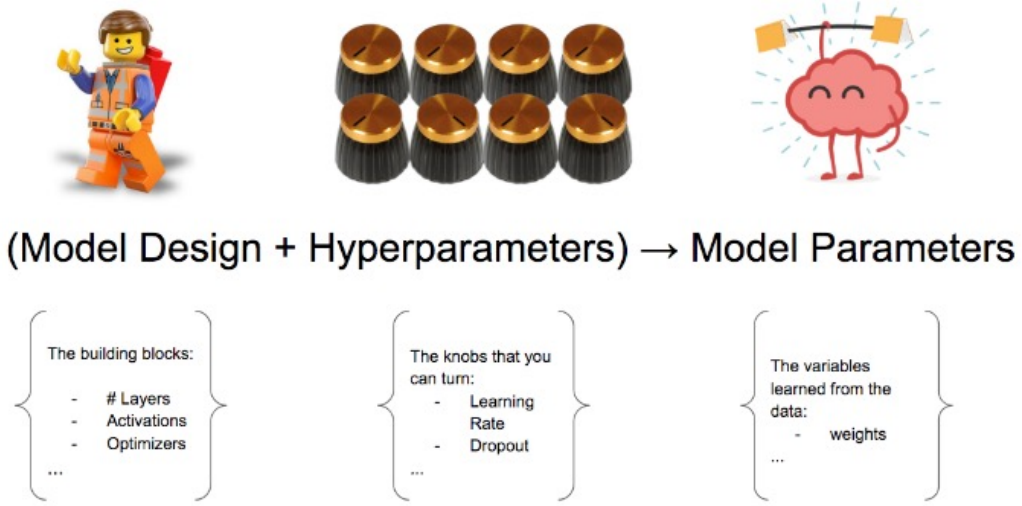
CS 505: Introduction to Natural Language Processing

Wayne Snyder
Boston University

Lecture 16 – Neural Network Tuning and Advanced Features



Lecture Plan

Best Practices and Advanced Features of Neural Networks for NLP

- Training vs. Retraining
- Avoiding recomputation by loading and saving
- Cross-Validation
- Optimizers and Learning Rates
- Regularization: L1, L2, Dropout
- Layer/Batch Normalization
- Early Stopping
- Batch Size (and an argument for asking chatGPT)

Initializing and Training Models

Simple question: What is the difference between initializing your model and training it in the same cell, or doing it in two different cells?

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device}")
print()

num_epochs = 500

spam_ham_model = SpamModel().to(device)         # <<====

training_losses = np.zeros(num_epochs)
val_losses      = np.zeros(num_epochs)

training_accuracy = np.zeros(num_epochs)
val_accuracy      = np.zeros(num_epochs)

loss_fn = nn.CrossEntropyLoss()
# optimizer = torch.optim.SGD(spam_ham_model.parameters(),lr=0.01)
# optimizer = torch.optim.Adam(spam_ham_model.parameters(),lr=0.001)
optimizer = torch.optim.Adagrad(spam_ham_model.parameters(),lr=0.01)
# optimizer = torch.optim.RMSprop(spam_ham_model.parameters(),lr=0.001

for epoch in tqdm(range(num_epochs)):
    # training
    spam_ham_model.train()
    t_loss = 0.0
    t_num_correct = 0

    for X_train_batch,Y_train_batch in spam_ham_training_dl:
        X_train_batch = X_train_batch.to(device)
        Y_train_batch = Y_train_batch.to(device)

        optimizer.zero_grad()
        Y_train_hat = spam_ham_model(X_train_batch)
        loss = loss_fn(Y_train_hat,Y_train_batch)
        loss.backward()
        optimizer.step()
        t_loss += loss.item()
```

```
spam_ham_model = SpamModel().to(device)
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device}")
print()

patience = 20     # how many epoches to wait with no improvement in
                  # loss score before termination

num_epochs = 1000

# retraining = True
retraining = False

# early_use_stopping = True
use_early_stopping = False

if not retraining:
  spam_ham_model = SpamModel().to(device)                    # <<====
  training_losses = np.zeros(num_epochs)
  val_losses      = np.zeros(num_epochs)

  training_accuracy = np.zeros(num_epochs)
  val_accuracy      = np.zeros(num_epochs)

loss_fn = nn.CrossEntropyLoss()
# optimizer = torch.optim.SGD(spam_ham_model.parameters(),lr=0.01)
# optimizer = torch.optim.Adam(spam_ham_model.parameters(),lr=0.001)
optimizer = torch.optim.Adagrad(spam_ham_model.parameters(),lr=0.01)
# optimizer = torch.optim.RMSprop(spam_ham_model.parameters(),lr=0.001)



for epoch in tqdm(range(num_epochs)):
    # training
    spam_ham_model.train()
    t_loss = 0.0
    t_num_correct = 0

    for X_train_batch,Y_train_batch in spam_ham_training_dl:
        X_train_batch = X_train_batch.to(device)                 # <<===
        Y_train_batch = Y_train_batch.to(device)                # <<====
```

Initializing and Training Models

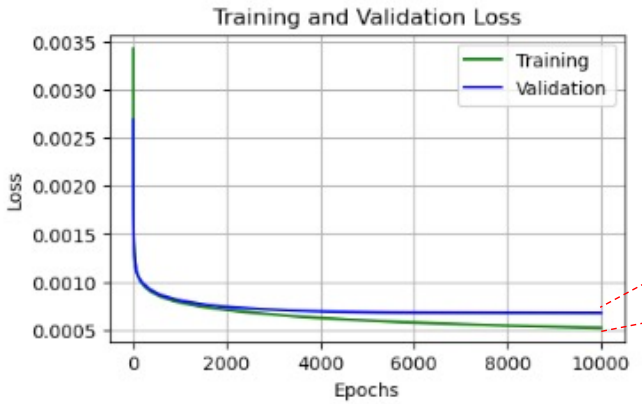
Simple question: What is the difference between initializing your model and training it in the same cell, or doing it in two different cells?

Answer: Nothing, as long as you always remember to run both cells for each training, especially if you change the hyperparameters!

What if you don't?

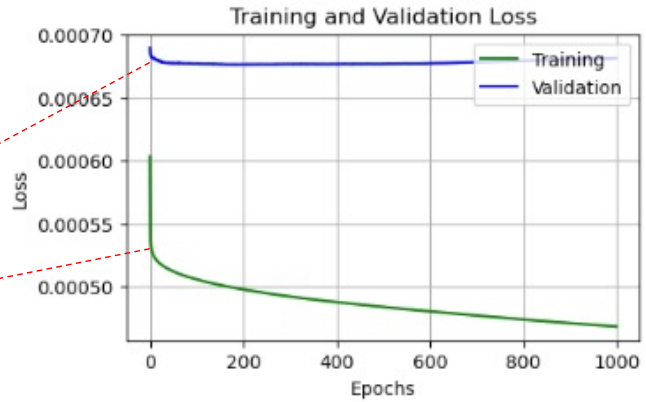
Then you will be retraining an already-trained model!
Fine if that is what you intend, but easy to forget, and you will get strange results:

Training for 10,000 epochs




```
Least Validation Loss 0.000675 found at epoch 9996.

Final Training Loss:   0.000521
Final Validation Loss: 0.000675
```

Retraining for 1000 more




```
Least Validation Loss 0.000676 found at epoch 178.

Final Training Loss:   0.000468
Final Validation Loss: 0.000681
```

Initializing and Training Models

Here is a nice way to avoid confusion and have both alternatives:

```
:
  1 # test if GPU is available
  2
  3 device = "cuda" if torch.cuda.is_available() else "cpu"
  4 print(f"Using {device}")
  5 print()
  6
  7 num_epochs = 10000
  8
  9 # Normally, will create model and train it in one run
 10 # If want to retrain the model with more epoches, set next to True
 11
 12 # retrain = True
 13 retrain = False
 14
 15 if not retrain:
 16     spam_ham_model = SpamModel().to(device)
 17     train_loss     = np.zeros(num_epochs)
 18     val_loss       = np.zeros(num_epochs)
 19
 20     train_accuracy = np.zeros(num_epochs)
 21     val_accuracy   = np.zeros(num_epochs)
 22
 23 learning_rate = 0.1
 24 learning_rate = 0.01
 25 # learning_rate = 0.001
 26 # learning_rate = 0.0001
 27
 28 # optimizer = torch.optim.SGD(spam_ham_model.parameters(),lr=learning_rate)
 29 # optimizer = torch.optim.Adam(spam_ham_model.parameters(),lr=learning_rate)
 30 optimizer = torch.optim.Adagrad(spam_ham_model.parameters(),lr=learning_rate)
 31 # optimizer = torch.optim.RMSprop(spam_ham_model.parameters(),lr=learning_rate)
```

Avoiding Redundant Computations

Try to avoid redoing the same expensive operations over and over!

We saw this with the Brown Corpus, which downloads the first time to your local disk, and thereafter checks to see if you already have it:

In [2]:
```
import numpy as np
import nltk
# First time you will need to download the corpus:
# Run the following and download the book collection

#nltk.download_shell()
```

In [3]:
```
from nltk.corpus import brown
nltk.download('brown')
```

```
[nltk_data] Downloading package brown to
[nltk_data]     /Users/waynesnyder/nltk_data...
[nltk_data]   Package brown is already up-to-date!
```

Out[3]: True

Avoiding Redundant Computations

You can do this with any data structure, such as tensors or numpy arrays.

Here is a way to do that with HW 04, Problem 2:

```
def load_glove_model(file):
   ...

import os

if os.path.exists(data_dir+'texts_vector.pt'):
    texts_vector = torch.load(data_dir+'texts_vector.pt')
else:
    glove_model = load_glove_model(data_dir+'glove.6B/glove.6B.100d.txt')
    sp = spacy.load('en_core_web_sm')
    emails_raw = pd.read_csv(data_dir+'data_pa5/enron_spam_ham.csv').to_numpy()
    texts_vector = []

    for text,label in tqdm(emails_raw):
        text_vector = torch.tensor([0]*100,dtype=torch.float32) #size of the word vector
        document=sp(text.lower())
        count = 0

        for word in document:
            if str(word) in glove_model:
                str_word    = str(word)
                text_vector = text_vector + glove_model[str_word]
                count += 1

        if count>0:
            text_vector /= count

        texts_vector.append((text_vector,torch.tensor(label,dtype=torch.int64)))

    torch.save(texts_vector,data_dir+'texts_vector.pt')
```

Thereafter, if you already have the file, just read it in using `torch.load(…)`

The first time, do the expensive computation, and save it to disk using `torch.save(….)`

Avoiding Redundant Computations

And of course you can save already-trained models, and not have to retrain them to use them later; again, we did this in HW 04 Problem 2:

```
    # now save the best model found so far, defined by least validation loss

    if epoch == 0:
        least_val_loss             = val_loss[epoch]

    if val_loss[epoch] < least_val_loss:
        least_val_loss         = val_loss[epoch]
        best_model             = copy.deepcopy(spam_ham_model)    # make deep copy to avoid I/O cost each time
        best_epoch             = epoch

 .....

#  testing using the best model found during training

best_model.eval()

testing_num_correct = 0

for X_test_batch,Y_test_batch in spam_ham_test_dl:
    X_test_batch = X_test_batch.to(device)
    Y_test_batch = Y_test_batch.to(device)

    Y_hat_test = best_model(X_test_batch)

    testing_num_correct += (torch.argmax(Y_hat_test,dim=1) == Y_test_batch).float().sum()

test_accuracy = testing_num_correct / N_test

 .....


    # Save the best model found during this training

    torch.save(best_model, data_dir+'best_spam_ham_model.pt')


    # You can load it any time to use it:

    spam_ham_model = torch.load(data_dir+'best_spam_ham_model.pt')
```

Use `copy.deepcopy(...)` to avoid doing expensive I/O in the main training loop!

Use the best model found during validation to do the testing!

Save and load the model as you wish!

Cross-Validation

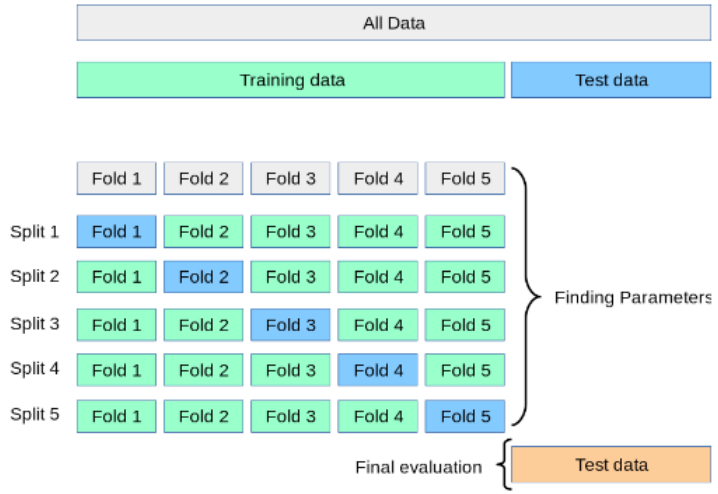
Cross-Validation is a dynamic alternative to choosing a fixed validation set:

- Split the data into a training set and a testing test, and hold out the testing set as usual;
- Now split the training set into K parts ("folds") of approximately equal size;
- Training occurs in cycles of K epochs:

For each k in range(K):

Train on all Folds *except* Fold k;

Validate on Fold k



- At the conclusion of cycle of K epochs, take the mean of the loss and accuracy metrics;
- Report performance metrics for these means every K epochs.

Cross-Validation

Cross-Validation and Static Validation have symmetric advantages and disadvantages:

Static validation is

- Simpler and faster;
- Very dependent on quality of split, especially for small or unbalanced data sets:
  - May overfit on that specific set;
  - Performance metrics may be skewed.

Cross-validation is

- More complex to implement, less efficient;
- Uses entire training set for validation, so exact split is less critical;
  - Less possibility of overfitting;
  - More accurate performance metrics
- Does not well as work for time-series data sets (e.g., stock prices, weather)

Punchline: Static validation is fine for large datasets (always shuffle!!);
Cross-Validation should be used for small or unbalanced data sets.

Cross-Validation

In Pytorch, you can simply create K different DataLoaders, and DIY as just described; sklearn also has a popular library KFold to make it simple but inefficient:

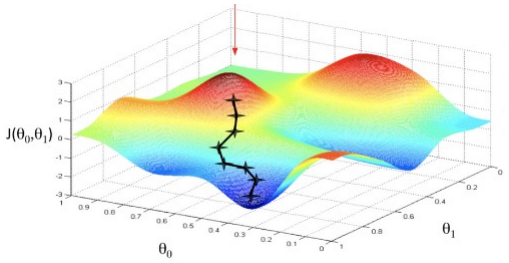
```
from sklearn.model_selection import KFold
from torch.utils.data import DataLoader, Subset

dataset = MyDataset()
k_fold = KFold(n_splits=5)
for train_indices, val_indices in k_fold.split(dataset):

    # Using Subset to create datasets for training and validation
    train_subset = Subset(dataset, train_indices)
    val_subset = Subset(dataset, val_indices)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32)

    # Now you can use train_loader and val_loader in your training and
validation loops
```

Optimizers: SGD, Adam, Adagram, RMSProp…

SGD (Stochastic Gradient Descent):

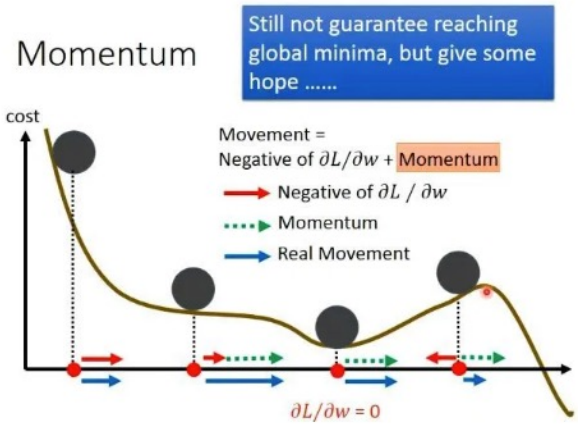
- Classic optimizer that updates the weights by taking a step in the direction of the negative gradient of the loss function

$$w_{\text{new}} = w_{\text{old}} - \eta \times \nabla_w \text{Loss}$$



Can set various parameters such as

- **Learning Rate Schedules** with
  - Step decay: reduce the learning rate by some factor each epoch
  - Exponential decay: Decrease the learning rate exponentially over the epochs;
  - 1/t decay: reduce the lr as the inverse of the square root of the number of epochs;
- **Momentum:**
  - Add a fraction (between 0 and 1) of the previous weight update to the current update
  - Helps accelerate in the relevant direction and dampen c
- **Weight decay:**
  - Equivalent to L2 regularization (later in the lecture)



Optimizers: SGD, Adam, Adagram, RMSProp…

SGD (Stochastic Gradient Descent):

- Classic optimizer that updates the weights by taking a step in the direction of the negative gradient of the loss function

$$w_{\text{new}} = w_{\text{old}} - \eta \times \nabla_w \text{Loss}$$

Can set various p

Adagrad (Adaptive Gradient Algorithm):

- Stored a running sum of the squares of past gradients and divides the learning rate by the square root of the running sum:

$$w_{\text{new}} = w_{\text{old}} - \frac{\eta}{\sqrt{s_t + \epsilon}} \times \nabla_w \text{Loss} \qquad s_t = s_{t-1} + \nabla_w \text{Loss} \odot \nabla_w \text{Loss}$$

Pro: Adapts to size of gradients. Con: Can adapt too much and stop learning!

RMSProp (Root Mean Square Propagation):

- More effective version of Adagrad, using a moving average of squared past gradients:

$$s_t = \beta s_{t-1} + (1 - \beta) \nabla_w \text{Loss} \odot \nabla_w \text{Loss}$$

RMSProp tends to work better for very deep neural networks

Optimizers: SGD, Adam, Adagram, RMSProp…

Adam (Adaptive Moment Estimation):

Improves on Adagrad and RMSprob by combining both approaches with regard to past gradients:

- Keep a moving weighted average of both the past gradients and the squared past gradients (called first and second moments), and adjust the learning rate accordingly.
- Corrects for initial bias in the moving averages, so tends to have more stable starts than other algorithms.

Punchlines:

- SGD has many parameters which can be tuned for excellent performance, and may lead to better performance.
- Adam is the default optimizer for many tasks because it tends to “work well out of the box” without a lot of tuning.

Regularization: L1, L2, Dropout

Regularization attempts to prevent overfitting by preventing models from becoming too complex. There is a large variety of ways to accomplish this:

- **Adding noise:**
  - Produce random fluctuations in the data through augmentation;
  - The network generalizes instead of focusing on the details.
- **L1 Regularization (Lasso Regression):**
  - Adds a penalty proportional to the absolute value of the coefficients:

$$\lambda \sum |w_i|$$

  - This prevents the parameters from becoming too large, limiting their range.
- **L2 Regularization (Ridge Regression):**
  - Adds a penalty proportional to the square of the magnitude of the coefficients:

$$\lambda \sum w_i^2$$

L2 is generally preferred, since L1 can force some parameters to 0.

Regularization: L1, L2, Dropout

L2 Regularization is accomplished in Pytorch using the weight_decay parameter in the optimizer:

```
39 # weight_decay = 0.02 is the strength of the L2 regularization
40
41 optimizer = torch.optim.Adam(spam_ham_model.parameters(),lr=0.01, weight_decay = 0.02)
```

The effect is to add the following penalty term to the loss L calculated during training:

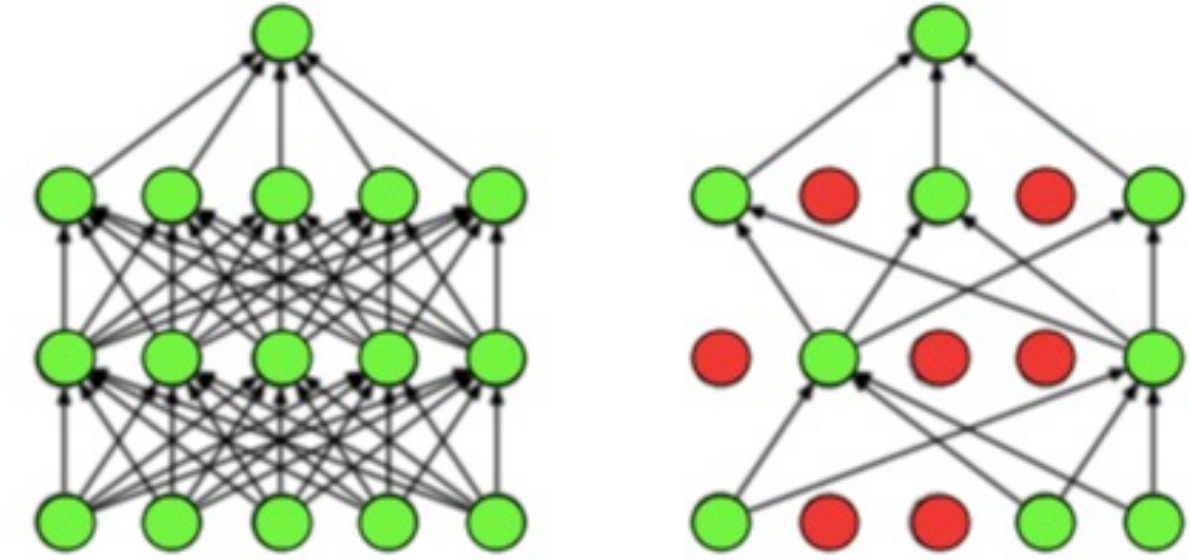
$$L' = L + \frac{\text{weight\_decay}}{2} \sum w^2$$

Parameters of model

(L1 regularization is not implemented in Pytorch and you would have to DIY.)

Regularization: L1, L2, Dropout

- **Dropout**
  - During training, parameters are set to 0 with some probability p
  - This prevents parameters from co-evolving and effectively memorizing the data
  - The knowledge implicit in the data is generalized throughout the network and not localized in specific parameters



(a) Standard Neural Net

(b) After applying dropout.

Note: Due to the random nature of dropout, different neurons will be dropped out for each data sample.

Regularization: L1, L2, Dropout

Dropout in Pytorch is easily accomplished with a dropout layer build into the network geometry:

```
:
1  class SpamModel(torch.nn.Module):
2      def __init__(self):
3          super(SpamModel, self).__init__()
4          self.linear1 = torch.nn.Linear(100,15)
5          self.activation1 = torch.nn.ReLU()
6          self.linear2 = torch.nn.Linear(15,2)
7          self.dropout = nn.Dropout(0.4)          # dropout neurons with probability 0.4
8
9      def forward(self, x):
10         x = self.linear1(x)
11         x = self.dropout(x)
12         x = self.activation1(x)
13         x = self.linear2(x)
14         return x
```

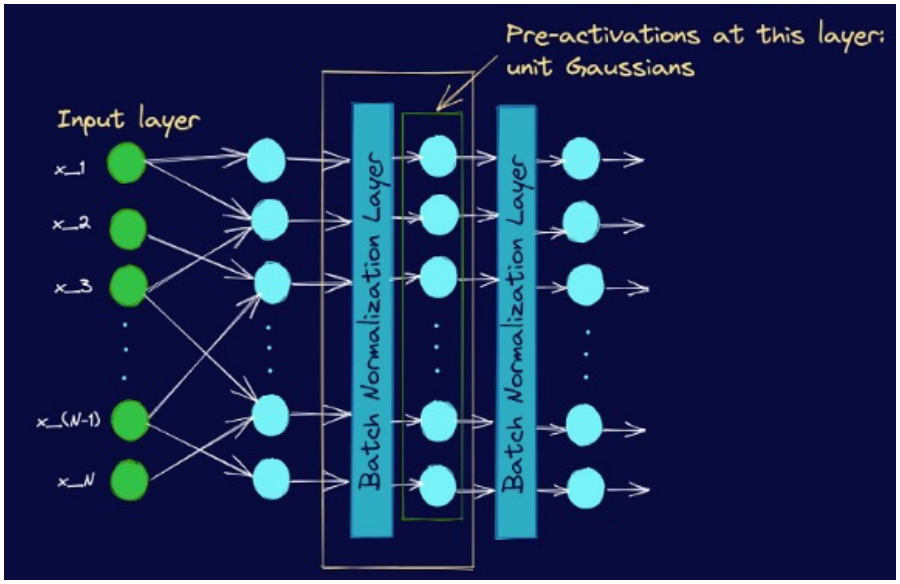
Layer and Batch Normalization

Layer Normalization: For each output value from a layer:

- Compute its mean μ and standard deviation σ ;
- Normalize to mean 0 and standard deviation 1; and then
- Scale and shift it by two parameters learned during training.

This is done after every individual data sample.

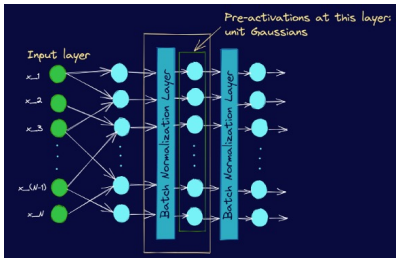
$$y_i = \gamma \left(\frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \right) + \beta$$



Batch Normalization is the same process, but applied to all layer outputs for a whole mini-batch.

Normalization is also considered to be a form of regularization, because it limits the range of parameters.

Layer and Batch Normalization

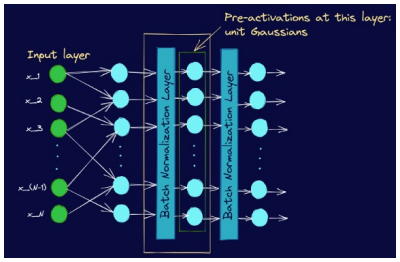


Why normalize layer and batch outputs?

- Helps gradient flow by avoiding disappearing or exploding gradients;
- Acts as a regularizer to avoid overfitting by introducing “useful noise” into the parameters;
- Smooths the gradient landscape:
  - Allows for higher learning rates and faster convergence;
  - Makes weight initialization strategy less critical.

Normalization is particularly effective with deep networks.

Layer and Batch Normalization



Batch Normalization in Pytorch:

```
class BatchNormNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(BatchNormNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)              # note that normalization is
        x = F.relu(x)                # done before relu or sigmoid
        x = self.fc3(x)
        return x
```

Early Stopping

Your goal is to find the best possible model for your task, typically measured by

- Minimum loss score
- Maximum accuracy score
- Optimal value of some other metric (specificity, F1, etc.)

The consensus view is that “it depends” but as a first approximation, loss is better than accuracy.

There is no reason to overfit by continuing past this point, and there are simple ways to implement early stopping:

- Stop when you reach some threshold of loss;
- Define a parameter `patience`, and stop training if your model does not improve (i.e., the loss does not decrease) after `patience` epochs. Typical values are 20 – 50 epochs.

Early Stopping

Early Stopping in Pytorch:

```
8
9  patience    = 50      # how many epoches to wait with no improvement in
10                       # loss score before termination
11
12 early_use_stopping = True
13 # use_early_stopping = False
14
```

```
87    # now save the best model found so far, defined by least validation loss
88
89    if epoch == 0:
90        least_val_loss              = val_loss[epoch]
91        num_epoches_no_improvement = 0
92
93    if val_loss[epoch] < least_val_loss:
94        least_val_loss          = val_loss[epoch]
95        best_model              = copy.deepcopy(spam_ham_model)    # make deep copy to avoid I/O cost each time
96        best_epoch              = epoch
97        num_epoches_no_improvement = 0
98    else:
99        num_epoches_no_improvement += 1
100
101   last_epoch = epoch          # save if early stopping
102
103   # early stopping
104
105   if use_early_stopping and num_epoches_no_improvement == patience:
106       print(f"Early stopping at epoch {last_epoch}, no improvement in validation loss after {patience} epoches."
107       break
108
```

Effect of Batch Size

You can't do better in most cases than asking chatGPT to do the research and summarize findings on all these matters. Maybe it meditates on its own creation? Here is its excellent summary of the effect of batch size on training:

WA

What is the effect of batch size on training a neural network?

The batch size, which is the number of training examples used in one forward-backward pass (one iteration), has a significant impact on the training dynamics and the performance of a neural network. Here are the effects and considerations associated with different batch sizes:

1. **Computational Efficiency**:
   - **Larger batches** can benefit from parallel processing capabilities of modern GPUs, leading to faster computation per sample.
   - **Smaller batches** might not fully utilize the computational resources, especially on powerful GPUs.
2. **Memory Usage**:
   - **Larger batches** consume more memory on the GPU, which might limit the model size or the maximum feasible batch size.
   - **Smaller batches** are more memory-efficient.

Effect of Batch Size

3. **Convergence Speed**:
   - **Larger batches** provide a more accurate estimate of the gradient. Fewer iterations per epoch may lead to faster convergence in terms of epochs but not necessarily in terms of wall-clock time.
   - **Smaller batches** introduce noise in the gradient estimate, which can have a regularizing effect and might help escape from local minima or saddle points. However, it might also lead to more oscillatory convergence trajectories.
4. **Generalization**:
   - There's empirical evidence suggesting that smaller batches can lead to models that generalize better, possibly due to the aforementioned noise acting as a form of implicit regularization.
   - Larger batches may converge to sharper minima, which might not generalize as well to the validation or test set. This has led to the idea that the noise in gradient estimates from smaller batches can be beneficial.
5. **Stability and Learning Rate**:
   - **Larger batches** often allow for larger learning rates as the gradient estimate is more accurate. However, the optimal learning rate typically grows sublinearly with the batch size.
   - **Smaller batches** might require smaller learning rates to maintain stable training.