Regularization and Optimization of DNNs

CSCI-P556 Applied Machine Learning
Lecture 16

D.S. Williamson

Agenda and Learning Outcomes

Today's Topics

- **Topics**:
  - Learning curves
  - Regularization and Optimization of DNNs
    - Vanishing and Exploding Gradients
    - Weight Initialization
    - Batch Normalization
    - Other Optimizers
    - Learning rate, dropout, …

DNN Training

Training Performance

- We can now begin the training process, since everything has been defined and initialized
- We need to do a few other things:
  - Keep track of training losses
  - Iterate over each epoch
  - Perform mini-batch gradient descent
  - Update the weights

```python
# Train the DNN
tr_avgLoss_list = []
tr_accuracy_list = []
dev_avgLoss_list = []
dev_accuracy_list = []

# Loop over epochs
for epoch in range(hyperparam.num_epochs):
    tr_num_correct = 0
    tr_num_samples = 0
    tr_total_loss = 0.0

    dev_num_correct = 0
    dev_num_samples = 0
    dev_total_loss = 0.0

    # Training
    dnn_model.train(True)

    with torch.set_grad_enabled(True):
        for local_batch, local_labels in training_gen:
            optimizer.zero_grad()

            local_batch = local_batch.float()
            local_labels = local_labels.float()
            local_batch, local_labels = Variable(local_batch), Variable(local_labels)

            # Model computations
            out1 = dnn_model(local_batch)

            #CrossEntropy loss calculation
            pLoss = loss(out1,local_labels.long())
            tr_total_loss += pLoss*hyperparam.bs #Correct for average based on batch size

            # Backpropagation
            pLoss.backward() #gradient calculation
            optimizer.step() #weight update


            sel_class = torch.argmax(out1,dim=1)

            tr_num_correct += sel_class.eq(local_labels).sum().item()
            tr_num_samples += hyperparam.bs

        tr_avgLoss = tr_total_loss/len(training_gen.dataset)
        tr_avgLoss_list.append(tr_avgLoss)

        tr_accuracy = tr_num_correct/tr_num_samples
        tr_accuracy_list.append(tr_accuracy)
```

DNN Training

Validation Loss for Performance Evaluation

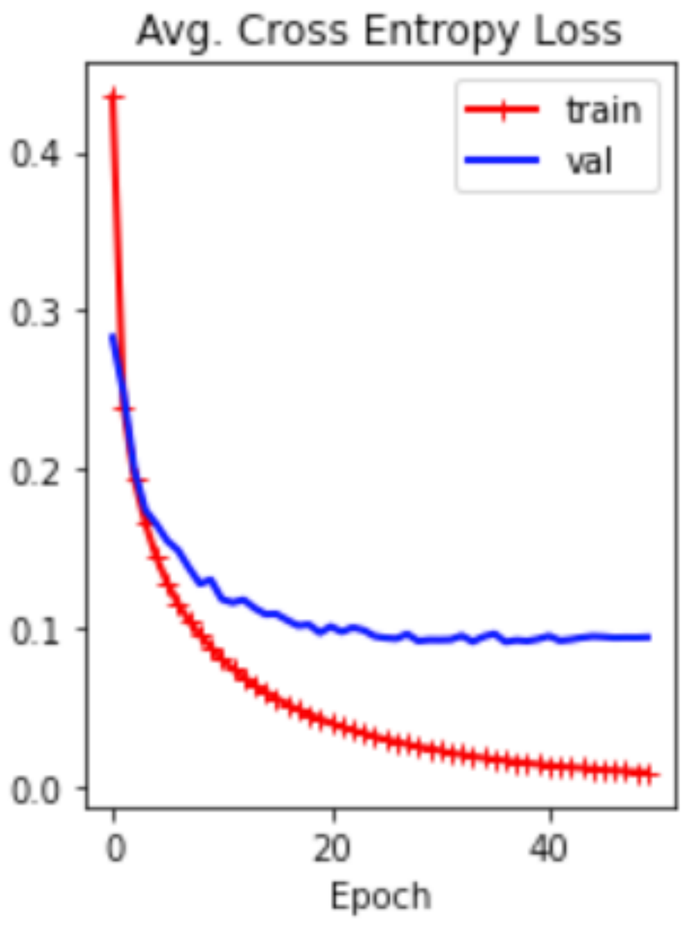
- We need to always be aware of the potential to overfit
- Hence, we should evaluate the model as it trains, using the validation/development data
- DO NOT update network based on this
  - Can use this for early stopping and model selection (more on this next)

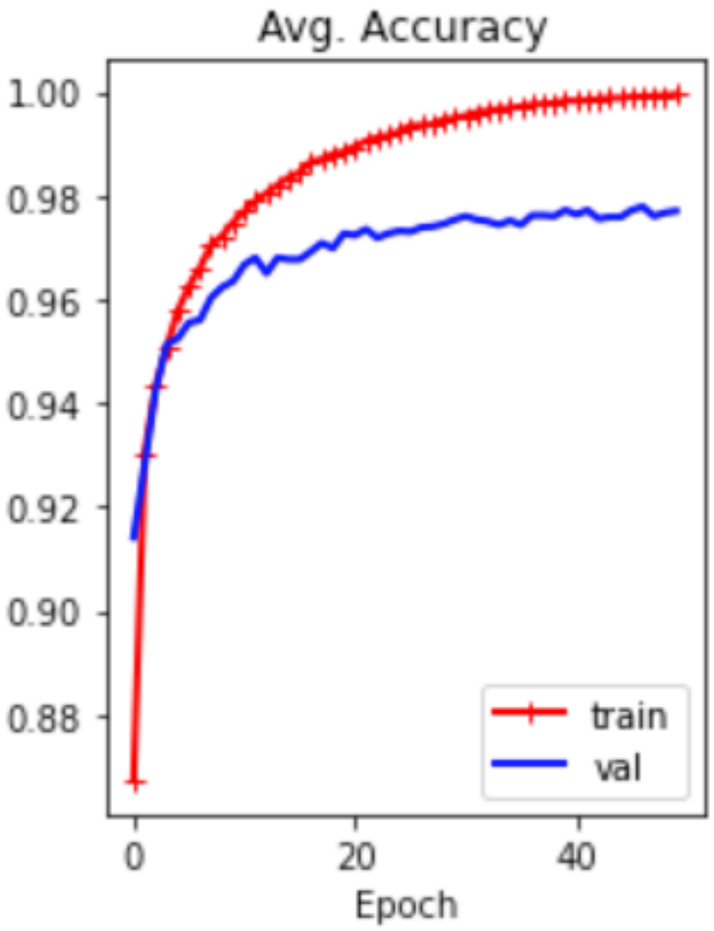
```
# Validation
with torch.set_grad_enabled(False):
    dnn_model.eval()

    for local_batch, local_labels in dev_gen:

        local_batch = local_batch.float()
        local_labels = local_labels.float()
        local_batch, local_labels = Variable(local_batch), Variable(local_labels)

        # Model computations
        out1 = dnn_model(local_batch)

        #CrossEntropy loss calculation
        pLoss = loss(out1,local_labels.long())
        dev_total_loss += pLoss*hyperparam.bs #Correct for average based on batch size

        sel_class = torch.argmax(out1,dim=1)

        dev_num_correct += sel_class.eq(local_labels).sum().item()
        #print(correction)
        dev_num_samples += hyperparam.bs

    dev_avgLoss = dev_total_loss/len(dev_gen.dataset)
    dev_avgLoss_list.append(dev_avgLoss)

    dev_accuracy = dev_num_correct/dev_num_samples
    dev_accuracy_list.append(dev_accuracy)
```

Evaluating Training and Validation Errors

Learning Curves

- Performance improves for each iteration
- Works better on training data, than validation/development data
- Possibly could run this for more epochs



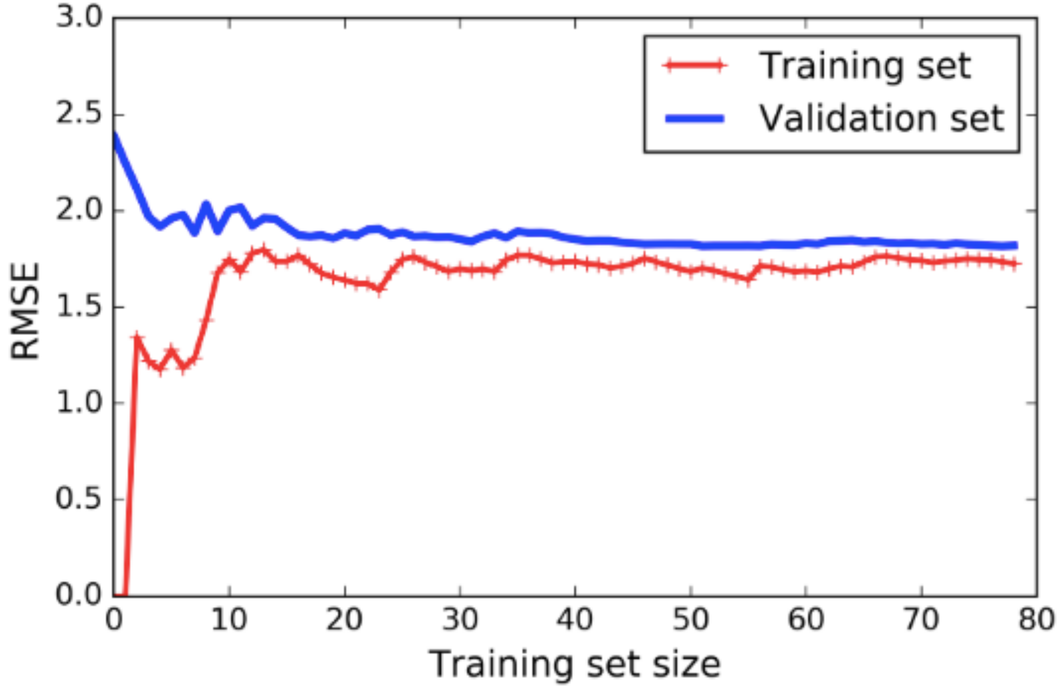


Determining Fit from Training/Val Errors

General way to determine if over- or under-fitting the data

- Learning curves allow us to evaluate the model's performance on the training and validation sets as a function of iteration (or a different hyper parameter).

- For the training data, the performance gets worse as the training set size increases, then it plateaus
- For the validation data, it doesn't generalize when the training set size is small, but it slightly improves as size increases (though not much)
- This is an example of ***underfitting***. The curves plateau, and they are close and high in value
  - Adding more training data doesn't help!
  - You need a more complex and better model

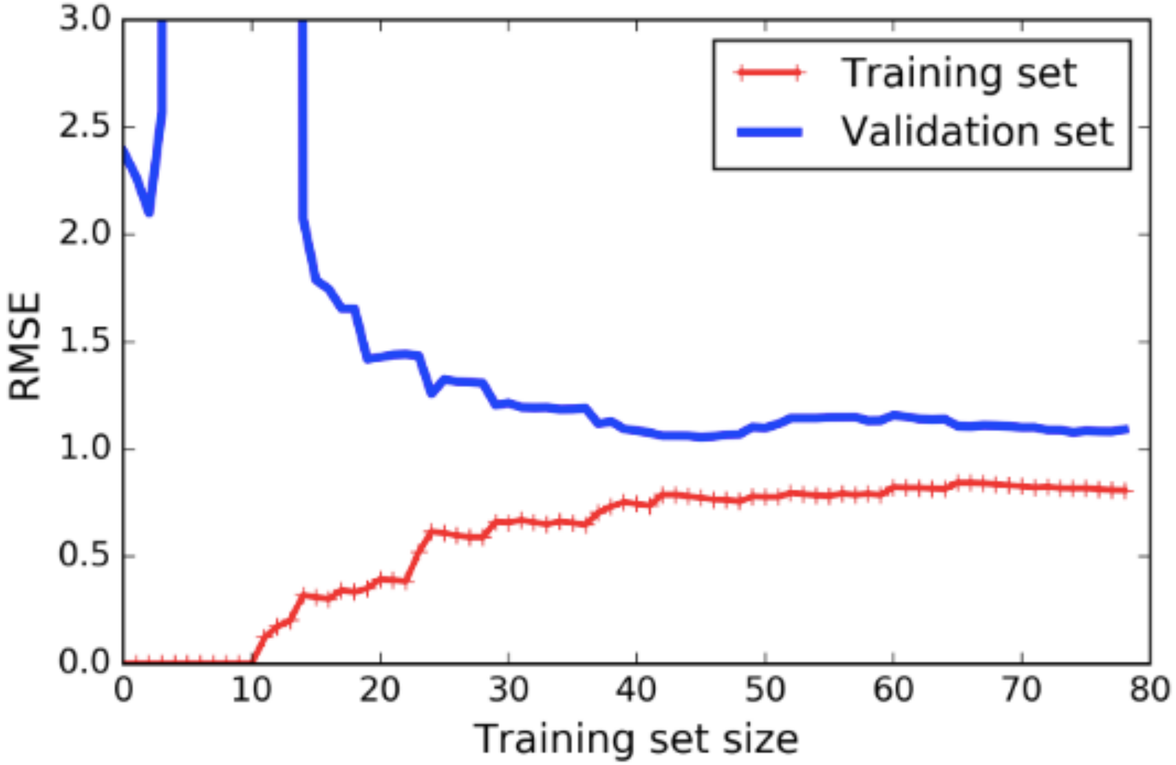


Determining Fit from Training/Val Errors

General way to determine if over- or under-fitting the data

- Let's suppose we increase the complexity of the model, and train this model using different training set sizes.

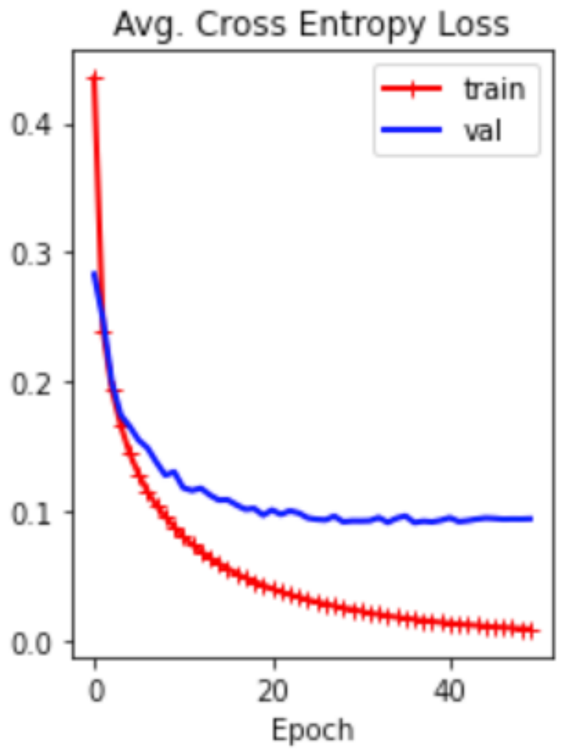
- These curves are similar but different from the more simpler model's curves
- The training curve is better and lower than the validation curve, with a noticeable gap between the two
  - This is a sign that the model is ***<u>overfitting.</u>***
  - Increasing the data set size may help avoid this, until the validation error reaches the training error

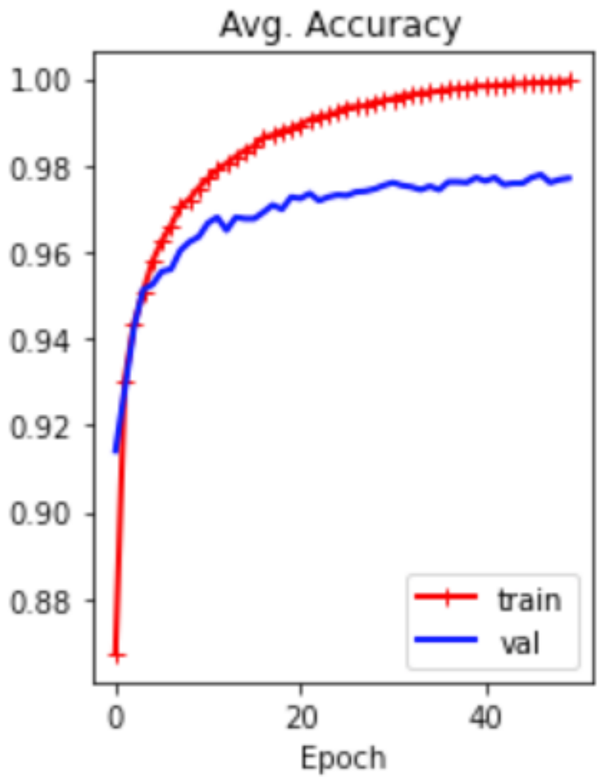


Evaluating Training and Validation Errors

Learning Curves

- Is this model ***overfitting*** or ***underfitting***?





The Bias-Variance Tradeoff

Generalization error

- A model's generalization error is based on:
  - **Bias**: wrong assumptions in the model. Leads to underfitting when is high.
  - **Variance***:* the model is overly sensitive to the data, leading to overfitting.
  - **Irreducible error**: the data is noisy. Need to get newer data or clean existing data in this case
- The ***tradeoff***:
  - Increasing complexity typically **increases** variance and **reduces** bias (hence overfitting)
  - Reducing a model's complexity **increases** its bias and **reduces** variance (hence underfitting)
- Regularization is subsequently used to reduce overfitting

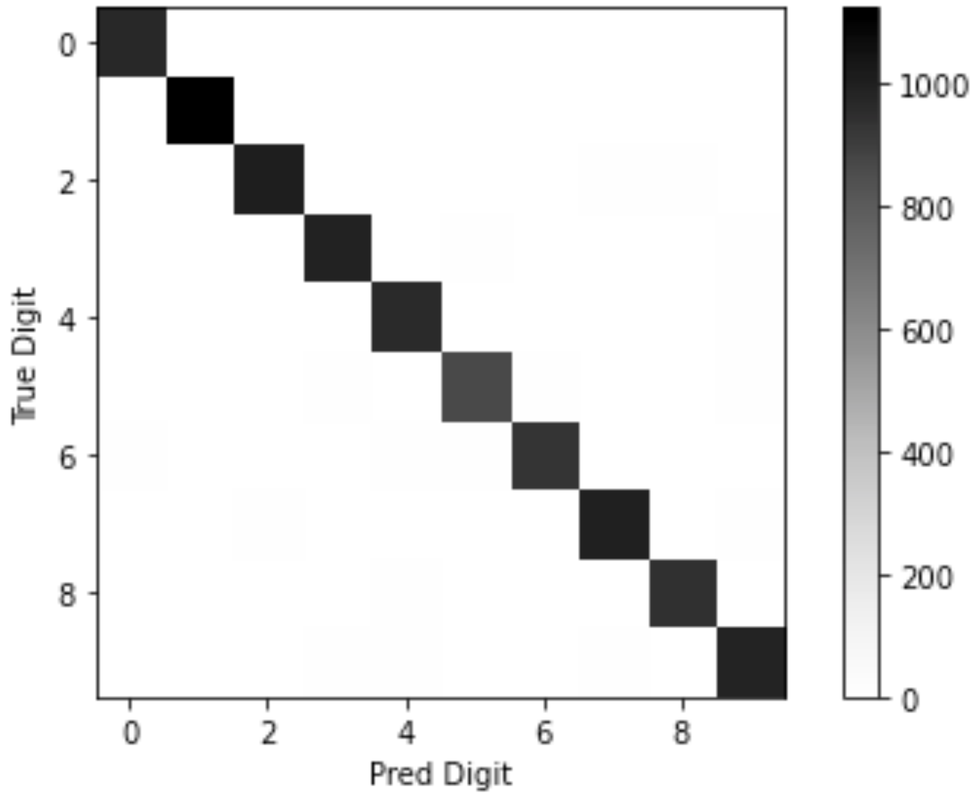
Test Performance for MNIST Digit Recognition

Generalization

```
# Testing
with torch.set_grad_enabled(False):
    dnn_model.eval()

    test_total_loss = 0.0
    test_num_samples = 0
    test_num_correct = 0

    pred = []
    y_testnew = []

    for local_batch, local_labels in testing_gen:

            local_batch = local_batch.float()
            local_labels = local_labels.float()
            local_batch, local_labels = Variable(local_batch), Variable(local_labels)

            # Model computations
            out1 = dnn_model(local_batch)


            #CrossEntropy loss calculation
            pLoss = loss(out1,local_labels.long())
            test_total_loss += pLoss*len(local_labels) #Correct for average based on batch

            sel_class = torch.argmax(out1,dim=1)
            pred += sel_class.tolist()
            y_testnew += local_labels.tolist()

            test_num_correct += accuracy_score(local_labels,sel_class,normalize=False)#sel_
            test_num_samples += len(local_labels)


    test_avgLoss = test_total_loss/len(testing_gen.dataset)
    test_accuracy = test_num_correct/test_num_samples

    print('Test Loss: {:>.9f}, Test Accuracy: {:>.5f}'.format(test_avgLoss,test_accuracy))
```

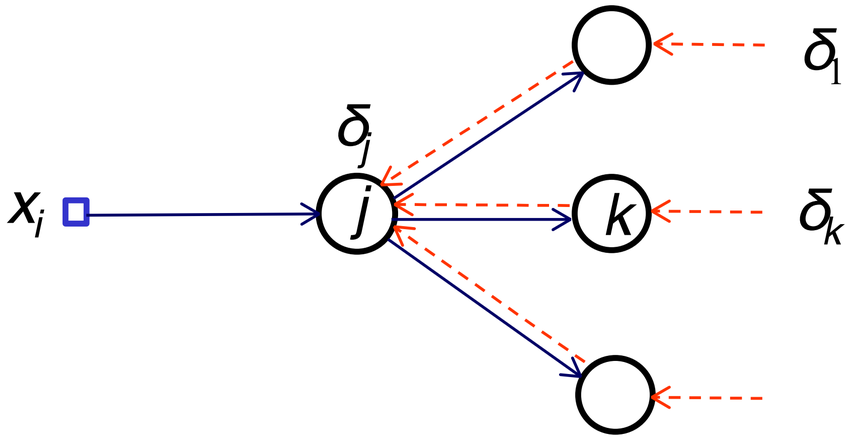
- **Results**

```
Test Loss: 0.082818724, Test Accuracy: 0.97620
```



Optimization: For Reducing Cost

Vanishing/Exploding Gradients Problems



- Backpropagation updates weights, starting with the output layer and going backwards towards the input layer, where the error gradient is propagated to update weights.
- There are two problems with this:
  - Gradients may get smaller as the propagation gets closer to the input layer, causing the early layer weights to not get updated and training never converges. This is called the ***vanishing gradients problem***.
  - Alternatively, the gradients may get bigger, so many layers get updated too much, where the algorithm diverges as well. This is the ***exploding gradients problem.***
- In 2010, Xavier Glorot and Yoshua Bengio showed that this is attributed to ***weight initialization*** and using a sigmoid activation function, which cause the variance of each output to increase (e.g. gradient becomes 0).

Network Weight Initialization

Xavier Initialization

- Proposed as an approach to alleviate vanishing/exploding gradient problem. Do not want the signal to explode or saturate.
- Authors argued that the ***variance of the outputs of each layer need to be equal to the variance of each layer's input***. The gradients also need to have equal variance before and after each layer.
- **Idea**: The connection weights of each layer must be initialized randomly based on one of the following probability distributions
  - Define fan_{avg} as the average of the number of inputs and the number of neurons. E.g. if the input has 784 attributes/dimensions and the first hidden layer has 300 neurons, then $fan_{avg} = (784 + 300)/2$
  - Option 1: Normal distribution with mean 0 and variance $\sigma^2 = 1/fan_{avg}$
  - Option 2: Uniform distribution between -r and +r, where $r = \sqrt{3/fan_{avg}}$

Network Weight Initialization

Other Initialization Strategies and activation functions

- Proposed initialization strategies for different types of activation functions (e.g. sigmoid activation function is not best - see 2010 paper by Glorot and Bengio)
  - Note that ReLU activation functions may cause neurons to “die” (always output 0). May use a ReLU variant (Leaky ReLU, Randomized leaky ReLU, Parametric leaky ReLU, and Exponential Linear Unit)

Initialization	**Activation Functions**	**Variance,**
Glorot	Linear, Tanh, Logistic, Softmax	$1/fan_{avg}$
He	ReLU & variants	$2/fan_{in}$
LeCun	SELU (Scaled ELU)	$1/fan_{in}$

Batch Normalization

Proper initialization and activation does not guarantee success

- The vanishing and exploding gradients may still occur, because the distributions between inputs and outputs changes during training, as the parameters change
- Another approach to address this is known as ***Batch Normalization***, which (1) zero-centers and normalizes the inputs of each layer, before the activation function is applied and (2) scales and shifts the result using two parameters for each layer.
- **Steps for Batch Normalization for a given layer:**
  - Compute sample mean and variance of the layer's input for a mini-batch
  - Subtract the mean from each sample in the batch, and divide by the standard deviation (e.g. make the inputs zero mean and unit variance)
  - Scale the normalized inputs by a scaling parameter (e.g. γ) and then add an offset (e.g. β)
  - During testing, use the mean and variance from the training set to normalize the testing data
- Overall, four parameters are learned (mean, variance, scale and offset)

Batch Normalization

Proper initialization and activation does not guarantee success

- Mathematically, batch normalization is performed as follows:

$$\mu_B = \frac{1}{N_B}\sum_{i=1}^{N_B} \mathbf{x}_i \qquad \sigma_B^2 = \frac{1}{N_B}\sum_{i=1}^{N_B} (\mathbf{x}_i - \mu_B)^2$$

Compute sample mean and variance of batch

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Normalize each input within the batch

$$\hat{\mathbf{x}}_i^{BN} = \gamma \hat{\mathbf{x}}_i + \beta$$

Further scale and shift the normalized input. This becomes the new input sample for this mini-batch

Batch Normalization In PyTorch

- Performed in-between layers

```
self.fc4 = nn.Linear(8*self.out_channel,self.linear_unit)
self.fc4_batchNorm = nn.BatchNorm1d(self.linear_unit)
```

Define function to perform batch normalization in __init()__ of DNN Module class

```
fc_out4 = self.fc4(out5)
#print(fc_out4.shape)
fc_out4 = self.fc4_batchNorm(fc_out4)
fc_out4 = self.leakyRelu(fc_out4)
```

Apply function in-between layers.

Impact of Batch Normalization

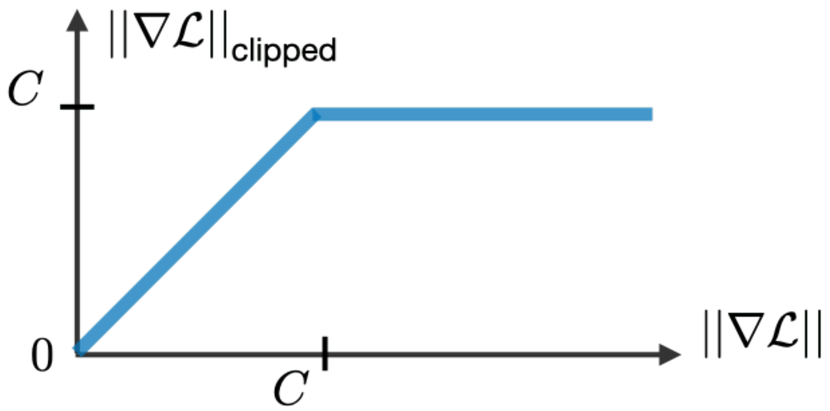
- **Pros**
  - Reduces vanishing gradient problem
  - Reduces network's sensitivity to initialization
  - Allows usage of larger learning rates (e.g. speeds-up convergence)
  - Reduces need for other regularization techniques
- **Cons**
  - Increases model complexity (e.g. more computations)
  - Network may become slower during testing/runtime

Gradient Clipping

How to deal with exploding gradients

- As stated previously, the gradient may become too large during training, which negatively impacts convergence
- To mitigate this, ***gradient clipping*** can be performed, which clips the gradient during back propagation so that they do not exceed a defined threshold.
  - Threshold is a tunable hyper parameter
  - Batch Normalization is preferred over this.

```
train_loss.backward() # Perform a backpropagation and calc
pt.nn.utils.clip_grad_value_(model.parameters(), 100)
optimizer.step() # Updates the weights accordingly to the
```



Faster Optimizers

Gradient Descent can be slow during Training

- **Momentum Optimization**: considers previous gradients during earlier iterations when updating the weights. It uses this to accelerate convergence.
  - Subtracts current gradient from a ***momentum vector, m.*** Then uses this new momentum vector to update the weights
  - β is a hyperparameter that controls acceleration. It is called ***momentum***, and it has values between 0 (no momentum) and 1. (Value of 0.9 indicates momentum is 10x faster)

$$\mathbf{m} = \beta\mathbf{m} - \nabla_\theta J(\theta) \qquad \theta = \theta + m$$

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Faster Optimizers

Other optimization approaches

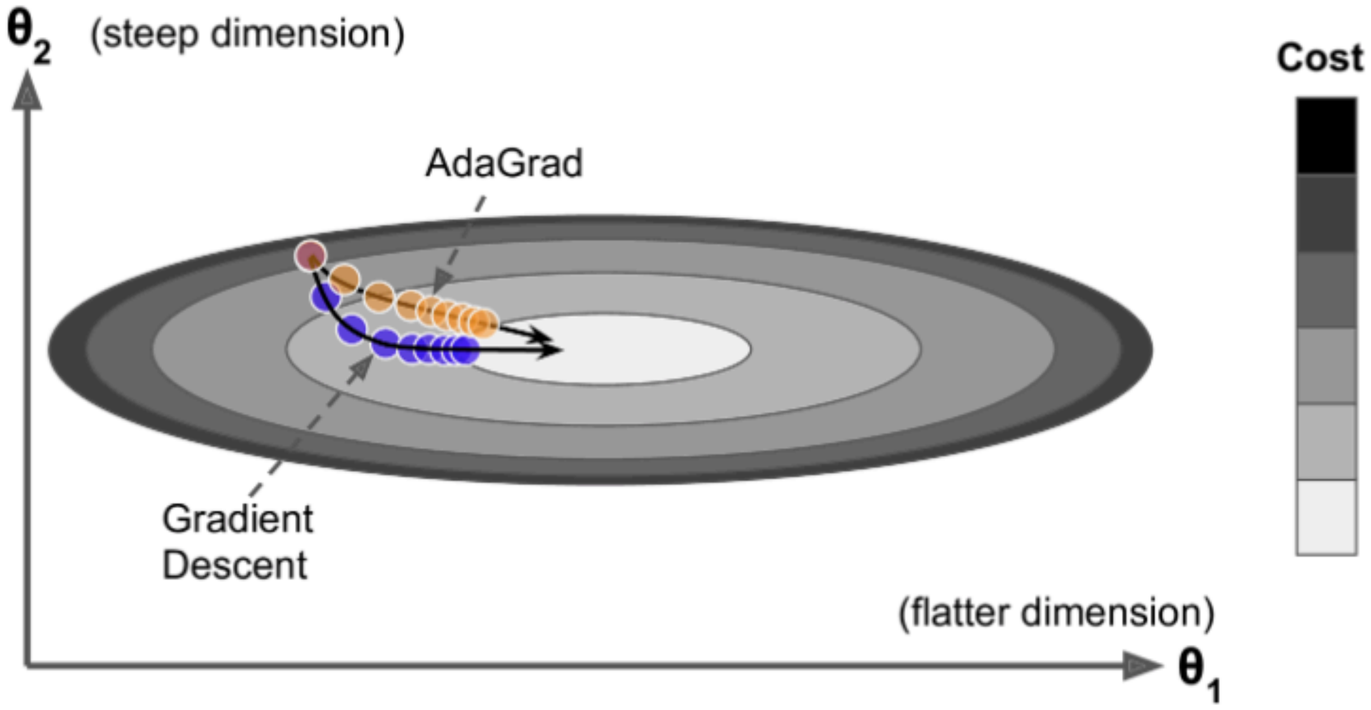
- **Nesterov Accelerated Gradient (NAG) Optimization**: generally faster than momentum. Measure the gradient of the cost function slightly ahead in the direction of momentum
  - Generally faster than momentum

$$\mathbf{m} = \beta\mathbf{m} - \nabla_\theta J(\theta + \beta\mathbf{m})$$

$$\theta = \theta + m$$

Faster Optimizers

Other optimization approaches



- **AdaGrad Optimization**: scale down the gradient vector along the steepest dimensions.
  - Hence, go down the flatter dimension first by pointing directly at the global optimum, instead of going down the steepest direction first (e.g. gradient descent).
  - Essentially, it decays the learning rate in an adaptive manner.
  - Unfortunately, training stops too early and it doesn't reach the global optimum for DNNs

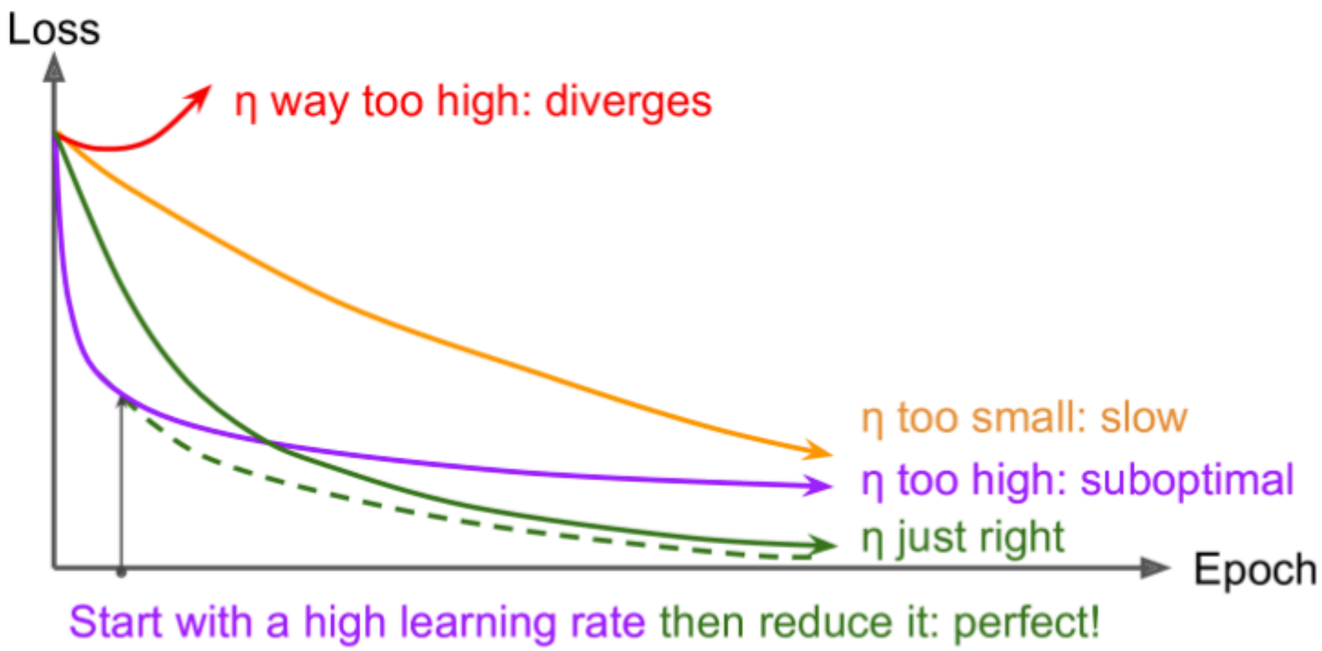
Faster Optimizers

Other optimization approaches

- **RMSProp Optimization**: Adagrad may not converge (learning rate becomes too small). RMSprop fixes this problem.
  - Accumulates only the gradients from the most recent iterations, using an exponential decay. Requires a decay rate hyper parameter
  - Generally better than AdaGrad
- **Adaptive Moment Estimation (Adam) Optimization**: combines momentum and RMSprop ideas
  - Often considered the “best” optimizer, but a separate study shows that Momentum or Nesterov may be better
  - Requires two momentum decay rate hyper parameters

Learning Rate Tuning

- The learning rate is a hyper-parameter that must be appropriately selected
  - When set to high, the solution never converges
  - When set to low, convergences takes too long
  - Ideally, the learning rate should cause the learning curve to quickly converge to a good solution
- The learning rate does not have to be constant, it can change over iterations. Not needed for AdaGrad, RMSProp, and Adam



Learning Rate Scheduling

Variable learning rates during training

- ***<u>Predetermined piecewise constant learning rate</u>***
  - <u>Idea</u>: Initialize the learning rate, then change it to a different value at a specific epoch
    - Ex. Set $\eta = 0.1$, then at the 50th epoch set $\eta = 0.0001$
    - Works, but need to determine appropriate values and when to change (e.g. more tuning)

Learning Rate Scheduling

Variable learning rates during training

- ***<u>Exponential Scheduling</u>***
  - <u>Idea</u>: Decay the learning rate by a factor γ, every N epochs. Hence, the learning rate changes with the epoch number.
    - Starts high then gets smaller to ensure convergence
    - $\eta(n) = \dfrac{\eta_0}{\gamma^{n/N}}$
    - Ex: For $\gamma = 10$, the learning rate will drop by a factor of 10 every N epochs
  - Note that PyTorch defines γ as $1/\gamma$ from the above definition. May also use *ExponentialLR()*, which assumes step of 1

```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

Learning Rate Scheduling

Variable learning rates during training

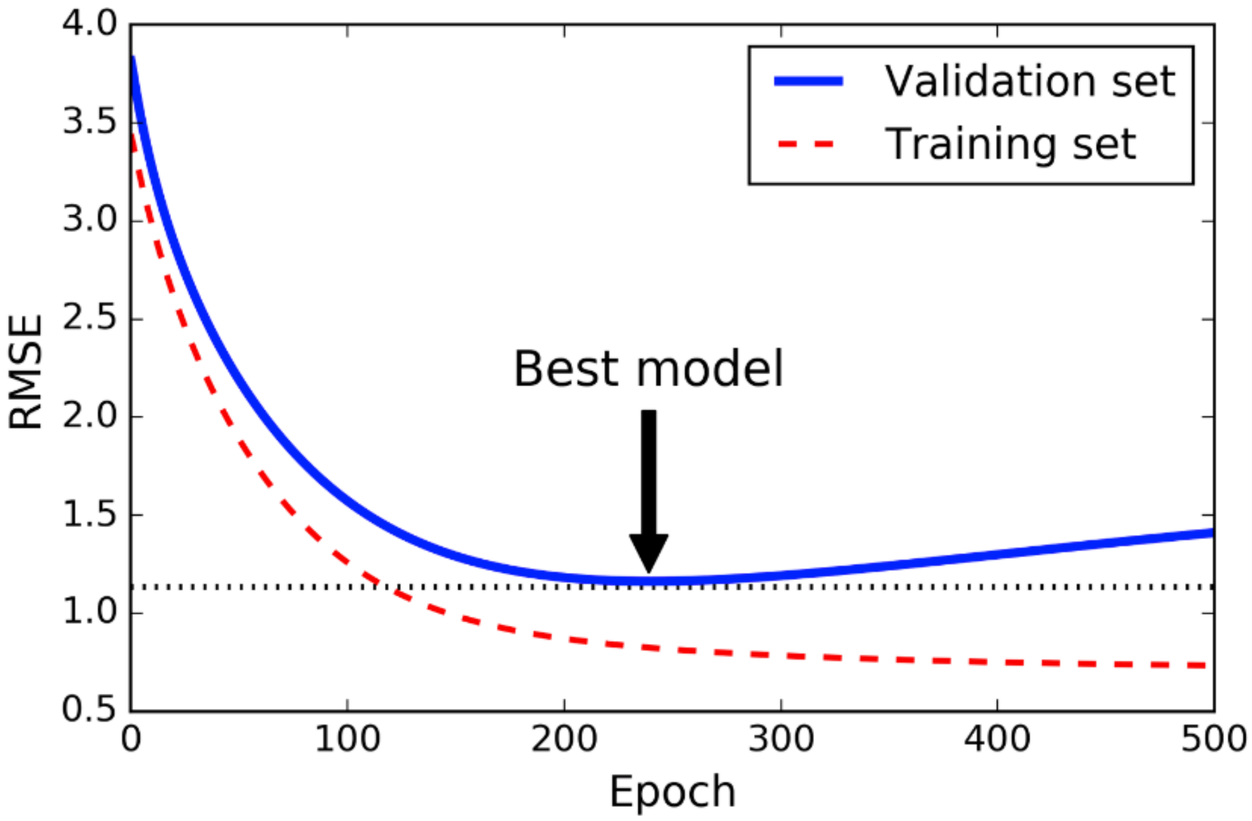
- ***Performance Scheduling***
  - Idea: Use the validation error to determine when to adjust the learning rate
    - Measure the validation error every N steps, and reduce by a factor of λ when the error stops dropping
    - See *ReduceLROnPlateau()* within *torch.optim.lr_scheduler*

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = ReduceLROnPlateau(optimizer, 'min')
>>> for epoch in range(10):
>>>     train(...)
>>>     val_loss = validate(...)
>>>     # Note that step should be called after validate()
>>>     scheduler.step(val_loss)
```

Regularization: For Improving Generalization

Early Stopping

- **Idea**: avoid overfitting, by stopping training when performance on the validation set starts getting worse
- Basic steps:
  - Evaluate model on validation set every N epochs
  - Save the model if performance is “better” than before
  - Count the number of steps since the last model was saved, and stop training when this number reaches a pre-defined limit (e.g. 3*N)
- Alternatively, keep track of the “best” performing model and run training for all epochs. Use the “best” performing model during testing. This is known as ***model selection***



L_1 and L_2 Regularization

Constrain values of weights during training

- **Idea**: Add a regularization term, $R(\theta)$, to the cost function (e.g. MSE) that is scaled by a factor, α, that controls how much you want to regularize

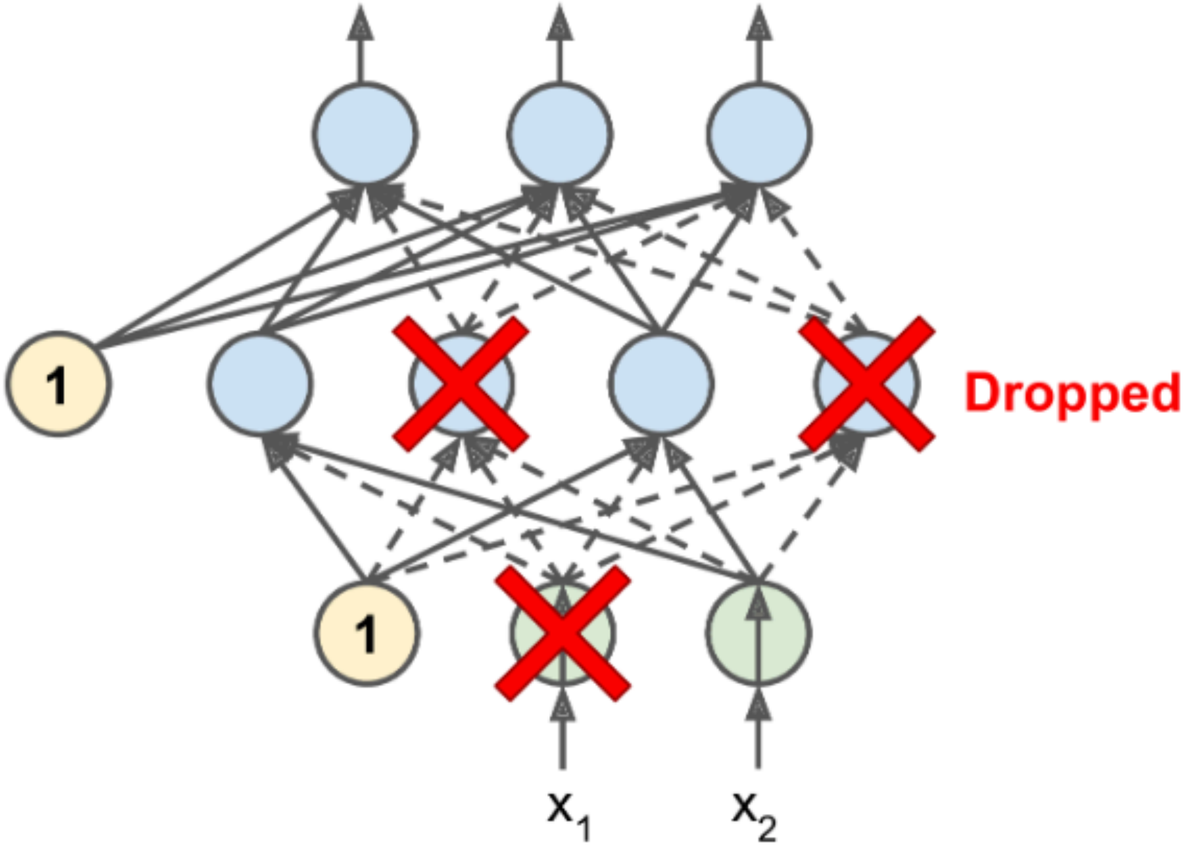
$$\text{Cost}(\theta) = MSE(\theta) + \alpha R(\theta)$$

  - **Ridge Regression** (L_2 Regularization): Goal is to keep model weights as small as possible. $R(\theta) = ||\theta||_2^2$
  - **Lasso Regression** (L_1 Regularization): Goal is to zero the weights of the least important features. $R(\theta) = \sum_{i=1}^{n} |\theta_i|$
  - **Elastic Net**: Mix of Ridge and Lasso regularization terms. $R(\theta) = rR_{ridge}(\theta) + \frac{1-r}{2} R_{LASSO}(\theta)$. Control mix with *r* (r = 0 means LASSO, r = 1 means Ridge).
- Regularization may be used with linear regression

Dropout

Avoid over-reliance on certain neurons. Improve adaptability

- Randomly ignore subset of neurons during training (excludes output layer)
- Every neuron has a probability of being dropped (outputs ignored) **during each training step.**
  - Controlled by drop out rate
  - Perform “coin flip” for each neuron to determine if it will be dropped
- May slow down convergence




```
torch.nn.Dropout(p=0.5, inplace=False)
```

Other Training Considerations

- **Reuse pre-trained network layers** (e.g. transfer learning) - speeds up training and requires less data
- **Freezing lower-layers of already-trained models** — may have detected low-level features for the problem
- **Unsupervised layer-by-layer Pre-training** — useful when don't have a lot of labeled data. Often uses auto encoders, but previously used Restricted Boltzmann Machines (RBM)

Next Class

Recurrent and Convolutional Neural Networks