CENG 783 – Deep Learning
Week 5: Artificial neural networks
Notes

- HW
- Project proposals
- Opportunity to work in a TUBITAK project
Today

- Representational capacity
- Overfitting, convergence, when to stop
- Data preprocessing and initialization
- Tips and tricks
- Final remarks
Representational capacity
Representational capacity

• Boolean functions:
  • Every Boolean function can be represented exactly by a neural network
  • The number of hidden layers might need to grow with the number of inputs

• Continuous functions:
  • Every bounded continuous function can be approximated with small error with two layers

• Arbitrary functions:
  • Three layers can approximate any arbitrary function

Universal approximation theorem:


Representational Capacity: Why go deeper if 3 layers is sufficient?

- Going deeper helps convergence in “big” problems.

- Going deeper in “old-fashion trained” ANNs does not help much in accuracy
  - However, with different training strategies or with Convolutional Networks, going deeper matters

Representational Capacity

- More hidden neurons $\Rightarrow$ capacity to represent more complex functions

- Problem: overfitting vs. generalization
  - We will discuss the different strategies to help here (L2 regularization, dropout, input noise, using a validation set etc.)
What the hidden units represent

**FIGURE 4.7**
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.
Number of hidden neurons

• Several rule of thumbs (Jeff Heaton)

  • The number of hidden neurons should be between the size of the input layer and the size of the output layer.

  • The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.

  • The number of hidden neurons should be less than twice the size of the input layer.
Number of hidden layers

- Depends on the nature of the problem
  - Linear classification? ➔ No hidden layers needed
  - Non-linear classification?
Model Complexity

- Models range in their flexibility to fit arbitrary data

Simple model
- Low bias
- Low variance
- Small capacity may prevent it from representing all structure in data

Complex model
- High bias
- High variance
- Large capacity may allow it to memorize data and fail to capture regularities

Slide Credit: Michael Mozer
Training Vs. Test Set Error

![Graph showing the relationship between model complexity and error for training and test sets. The graph illustrates the trade-off between model complexity and error, with an optimum model complexity point.](image)

Optimum Model Complexity

Error

Model Complexity

Training Set

Test Set

Slide Credit: Michael Mozer
Bias-Variance Trade Off

Error on Test Set vs. Model Complexity

- Underfit
- Optimum Model Complexity
- Overfit

Total Error

Bias^2

Variance

image credit: scott.fortmann-roe.com

Slide Credit: Michael Mozer
OVERFITTING, CONVERGENCE, AND WHEN TO STOP
Overfitting

• Occurs when training procedure fits not only regularities in training data but also noise.
  – Like memorizing the training examples instead of learning the statistical regularities
• Leads to poor performance on test set
• Most of the practical issues with neural nets involve avoiding overfitting
Avoiding Overfitting

• Increase training set size
  – Make sure effective size is growing; redundancy doesn’t help

• Incorporate domain-appropriate bias into model
  – Customize model to your problem

• Set hyperparameters of model
  – number of layers, number of hidden units per layer, connectivity, etc.

• Regularization techniques
  – “smoothing” to reduce model complexity
Incorporating Domain-Appropriate Bias Into Model

• Input representation
• Output representation
• Architecture
  – # layers, connectivity
  – e.g., convolutional nets, residual connections etc.
• Activation function
• Error function

Slide Credit: Michael Mozer
Customizing Networks

• Neural nets can be customized based on the problem domain
  – choice of error function
  – choice of activation function

• Domain knowledge can be used to impose domain-appropriate bias on model
  – bias is good if it reflects properties of the data set
  – bias is harmful if it conflicts with properties of data
Adding bias into a model

- Adding hidden layers or direct connections based on the problem

---

Direct I/O connections to learn easy parts of task

- NetTalk performs at about 70% without hidden units (guess)
- Performance up to 100% with hidden units
- Hidden units useful for handling exceptions
- E.g., XOR

---

Hidden units discover higher order features critical to performance (here, A & B)

---

Slide Credit: Michael Mozer
Adding bias into a model

- Modular architectures
  - Specialized hidden units for special problems
Adding bias into a model

• Local or specialized receptive fields
  – E.g., in CNNs
• Constraints on activities
• Constraints on weights

Constraints on activities
E.g. reduce amount of information flowing through net by encouraging binary-valued hidden units

\[ E = \sum_p (d_p - o_p)^2 + \sum_{h \text{hidden}} \alpha_h (1 - o_h) \]

Constraints among weights
E.g., T-C problem: Each hidden unit should detect the same feature, but shifted in position

Set \[ w_1 = w_2 \text{ initially} \]
\[ \Delta w_1 = \Delta w_2 = -\varepsilon (d - o) \sum_o \]
Adding bias into a model

• Use different error functions (e.g., cross-entropy)
• Use specialized activation functions
Adding bias into a model

- Introduce other parameters
  - Temperature
  - Saliency of input
Regularization

• Regularization strength can affect overfitting

\[ \frac{1}{2} \lambda w^2 \]
Regularization

• L2 regularization: $\frac{1}{2} \lambda w^2$
  – Very common
  – Penalizes peaky weight vector, prefers diffuse weight vectors
• L1 regularization: $\lambda |w|$
  – Enforces sparsity (some weights become zero)
  – Leads to input selection (makes it noise robust)
  – Use it if you require sparsity / feature selection
• Can be combined: $\lambda_1 |w| + \lambda_2 w^2$
• Regularization is not performed on the bias; it seems to make no significant difference
L1 vs. L2 optimization

\[ \min \|x\|_1 \text{ subject to } Ax = b \]

Harder to solve since it is not smooth. Single L1 solution.

\[ \min \|x\|_2 \text{ subject to } Ax = b \]

\[ \mathcal{L}(x) = \|x\|_2^2 + \lambda^T (Ax - b) \]
L0 regularization

• $L_0 = \left(\sum_i x_i^0\right)^{1/0}$
• How to compute the zeroth power and zeroth-root?
• Mathematicians approximate this as:
  – $L_0 = \#\{i \mid x_i \neq 0\}$
  – The cardinality of non-zero elements
• This is a strong enforcement of sparsity.
• However, this is non-convex
  – L1 norm is the closest convex form
L1 penalty: The L1 penalty is generally used as a substitute for the L0 penalty, where the L0 "norm" is just the number of non-zero components of the vector. The L0 norm can't effectively be used in practice due to the time complexity of the combinatorial problem it creates, but the L1 norm serves as an adequate substitute. The gradient on the L1 norm is constant with respect to the magnitude of each vector component, so all inputs are reduced equally. As an example, the L1 norm of (1, 3) is equal to the L1 norm of (0, 4). This forces most of the inputs to be extremely close to 0. So **use the L1 norm when you want sparsity.**

L2 penalty: The critical difference for the L2 norm is that the gradient is linear in the magnitude of each component of the vector. Thus, small values are favored, but its more favorable to decrease a large value than it is to make a small value even smaller because the gradient for small values is very small. As an example, the L2 norm of (1, 3) is root 10, and decreasing the first component by 1 results in a vector of (0, 3) with L2 norm of root(9) = 3 but decreasing the second component by 1 results in a vector of (1, 2) with L2 norm of root(5) < 3. Thus, its more favorable to decrease the larger components of the vector. Thus, **use the L2 norm when you don't want large activations.**

Hope this served as an alright answer. There are certainly people on here that know more about this than I do (and I'm not overly confident in the completeness of my answer), so I'd love to be corrected or for someone to tell me that I actually do understand the gist of this.
Regularization

• Enforce an upper bound on weights:
  – Max norm:
    • $\|w\|_2 < c$
    • Helps the gradient explosion problem
    • Improvements reported

• Dropout:
  – At each iteration, drop a number of neurons in the network
  – **Use a neuron’s activation** with probability $p$ (a hyperparameter)
  – Adds stochasticity!

Fig: Srivastava et al., 2014

Regularization: Dropout

• Feed-forward only on active units
• Can be trained using SGD with mini-batch
  – Back propagate only “active” units.
• One issue:
  – Expected output $x$ with dropout:
  – $E[x] = px + (1 - p)0$
• To have the same scale at testing time (no dropout), multiply test-time activations with $p$.

Fig: Srivastava et al., 2014
Regularization: Dropout

Training-time:

```python
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p  # first dropout mask
H1 *= U1  # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p  # second dropout mask
H2 *= U2  # drop!
out = np.dot(W3, H2) + b3
```

Test-time:

```python
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
out = np.dot(W3, H2) + b3
```
Regularization: **Inverted Dropout**

Perform scaling while dropping at training time!

**Training-time:**

```python
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p  # first dropout mask. Notice /p!
H1 *= U1  # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p  # second dropout mask. Notice /p!
H2 *= U2  # drop!
out = np.dot(W3, H2) + b3
```

**Test-time:**

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)  # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

Data Augmentation

Regularization Summary

• L2 regularization
• Inverted dropout with $p = 0.5$ (tunable)
When To Stop Training

1. Train $n$ epochs; lower learning rate; train $m$ epochs
   - bad idea: can’t assume one-size-fits-all approach

2. Error-change criterion
   - stop when error isn’t dropping
   - recommendation: criterion based on % drop over a window of, say, 10 epochs
     - 1 epoch is too noisy
     - absolute error criterion is too problem dependent
   - Another idea: train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
When To Stop Training

• 3. Weight-change criterion
  – Compare weights at epochs \((t - 10)\) and \(t\) and test:
    \[
    \max_i \left| w_i^t - w_i^{t-10} \right| < q
    \]
  – Don’t base on length of overall weight change vector
  – Possibly express as a percentage of the weight
  – Be cautious: small weight changes at critical points can result in rapid drop in error
Training Vs. Test Set Error

![Graph showing the relationship between error and model complexity for training and test sets. The graph illustrates the concept of overfitting and underfitting, with the optimum model complexity indicated by a vertical line.]
DATA PREPROCESSING AND WEIGHT INITIALIZATION
Data Preprocessing

• Mean subtraction
• Normalization
• PCA and whitening
Data Preprocessing: Mean subtraction

- Subtract the mean for each dimension:
  \[ x'_i = x_i - \hat{x}_i \]
- Effect: Move the data center (mean) to coordinate center
Data Preprocessing:
Normalization (or conditioning)

• Necessary if you believe that your dimensions have different scales
  – Might need to reduce this to give equal importance to each dimension
• Normalize each dimension by its std. dev. after mean subtraction:
  \[ x_i' = x_i - \mu_i \]
  \[ x_i'' = \frac{x_i'}{\sigma_i} \]
• Effect: Make the dimensions have the same scale

Data Preprocessing:
Principle Component Analysis

• First center the data
• Find the eigenvectors $e_1, ..., e_n$
• Project the data onto the eigenvectors:
  \[ x_i^R = x_i \cdot [e_1, ..., e_n] \]
• This corresponds to rotating the data to have the eigenvectors as the axes
• If you take the first $M$ eigenvectors, it corresponds to dimensionality reduction

Reminder: PCA

• Principle axes are the eigenvectors of the covariance matrix:

\[
\Sigma = \begin{bmatrix}
E[(X_1 - \mu_1)(X_1 - \mu_1)] & E[(X_1 - \mu_1)(X_2 - \mu_2)] & \cdots & E[(X_1 - \mu_1)(X_n - \mu_n)] \\
E[(X_2 - \mu_2)(X_1 - \mu_1)] & E[(X_2 - \mu_2)(X_2 - \mu_2)] & \cdots & E[(X_2 - \mu_2)(X_n - \mu_n)] \\
\vdots & \vdots & \ddots & \vdots \\
E[(X_n - \mu_n)(X_1 - \mu_1)] & E[(X_n - \mu_n)(X_2 - \mu_2)] & \cdots & E[(X_n - \mu_n)(X_n - \mu_n)]
\end{bmatrix}
\]
Data Preprocessing: Whitening

- Normalize the scale with the norm of the eigenvalue:
  \[ x_i^w = x_i^R / (\left[ \lambda_1, \ldots, \lambda_n \right] + \epsilon) \]
- \( \epsilon \): a very small number to avoid division by zero
- This stretches each dimension to have the same scale.
- Side effect: this may exaggerate noise.
Data Preprocessing: Example
Data Preprocessing: Summary

• We mostly don’t use PCA or whitening
  – They are computationally very expensive
  – Whitening has side effects
• It is quite crucial and common to zero-center the data
• Most of the time, we see normalization with the std. deviation
Weight Initialization

• Zero weights
  – Wrong!
  – Leads to updating weights by the same amounts for every input
  – Symmetry!

• Initialize the weights randomly to small values:
  – Sample from a small range, e.g., Normal(0,0.01)
  – Don’t initialize too small

• The bias may be initialized to zero
  – For ReLU units, this may be a small number like 0.01.

Note: None of these provide guarantees. Moreover, there is no guarantee that one of these will always be better.
Initial Weight Normalization

- **Problem:**
  - Variance of the output changes with the number of inputs
  - If $s = \sum_i w_i x_i$:

  \[
  \text{Var}(s) = \text{Var}\left(\sum_i w_i x_i\right) \\
  = \sum_i \text{Var}(w_i x_i) \\
  = \sum_i \left[ E(w_i)^2 \text{Var}(x_i) + E(x_i)^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \right] \\
  = \sum_i \text{Var}(x_i) \text{Var}(w_i) \\
  = (n \text{Var}(w)) \text{Var}(x)
  \]

\[
\text{Var}(X) = E[(X - \mu)^2]
\]
Initial Weight Normalization

• **Solution:**
  – Get rid of $n$ in $\text{Var}(s) = (n \text{Var}(w))\text{Var}(x)$
  – How?

• $w_i = \text{rand}(0,1)/\sqrt{n}$
  – Why?
  – $\text{Var}(aX) = a^2 \text{Var}(X)$

• If the number of inputs & outputs are not fixed:
  – $w_i = \text{rand}(0,1) \times \frac{2}{\sqrt{n_{in}+n_{out}}}$

Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

**Alternative: Batch Normalization**

- **Normalization is differentiable**
  - So, make it part of the model (not only at the beginning)
  - I.e., perform normalization during every step of processing
- **More robust to initialization**
- **Issue: How to normalize at test time?**
  1. Store means and variances during training, or
  2. Calculate mean & variance over your test data

---

To sum up

• Initialization and normalization are crucial
• Different initialization & normalization strategies may be needed for different deep learning methods
  – E.g., in CNNs, normalization might be performed only on convolution etc.
• More on this later
LOSS FUNCTIONS, AGAIN
Loss functions

• **A single correct** label case (classification):
  
  – Hinge loss:
    
    \[ L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1) \]
  
  – Soft-max:
    
    \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) \]
Loss functions

• **Many correct** labels case:
  
  – Binary prediction for each label, independently:
    
    • $L_i = \sum_j \max(0, 1 - y_{ij}f_j)$
    
    • $y_{ij} = +1$ if example $i$ is labeled with label $j$; otherwise $y_{ij} = -1$.

  – Alternatively, train logistic loss for each label (0 or 1):
    
    $$L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$$
Loss functions

• Regression case (a continuous label):
  – L2 Norm squared (L2 Loss):
    • \( L_i = |f - y_i|^2 \)
    • \( \frac{\partial L_i}{\partial f_j} = f_j - (y_i)_j \)
  – L1 Norm:
    • \( L_i = |f - y_i|_1 = \sum_j |f_j - (y_i)_j| \)
    • \( \frac{\partial L_i}{\partial f_j} = \text{sign}(f_j - (y_i)_j) \)

Reminder:
\[ \|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}. \]
Mean-squared error Loss: Caution

• MSE loss asks for a more difficult constraint:
  – Learn to output a response that is exactly the same as the correct label
  – This is harder to train

• Compare, e.g., softmax:
  – Which asks only one response to be maximum than others.
Loss functions

• What if we want to predict a graph, tree etc.? Something that has structure.
  – **Structured loss**: formulate loss such that you minimize the distance to a correct structure
  – Not very common
ISSUES & PRACTICAL ADVICES
Issues & tricks

• Vanishing gradient
  – Saturated units block gradient propagation (why?)
  – A problem especially present in recurrent networks or networks with a lot of layers
• Overfitting
  – Drop-out, regularization and other tricks.
• Tricks:
  – Unsupervised pretraining
• Batch normalization (each unit’s preactivation is normalized)
  – Helps keeping the preactivation non-saturated
  – Do this for mini-batches (adds stochasticity)
  – Backprop needs to be updated
Unsupervised pretraining

Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left:** 1 hidden layer. **Right:** 4 hidden layers.

Why Does Unsupervised Pre-training Help Deep Learning?

Dumitru Erhan
Yoshua Bengio
Aaron Courville
Pierre-Antoine Manzagol
Pascal Vincent
Département d’informatique et de recherche opérationnelle
Université de Montréal
2920, chemin de la Tour
Montréal, Quebec, H3T 1J8, Canada

Sascha Bengio
Google Research
1600 Amphitheatre Parkway
Mountain View, CA, 94043, USA

BENGIO@GOOGLE.COM
Unsupervised pretraining

Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to zero, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments performed by Dumitru Erhan.
What if things are not working?

• Check your gradients by comparing them against numerical gradients
  – Check whether you are using an appropriate floating point representation
    • Be aware of floating point precision/loss problems
  – Turn off drop-out and other “extra” mechanisms during gradient check
  – This can be performed only on a few dimensions

• Regularization loss may dominate the data loss
  – First disable regularization loss & make sure data loss works
  – Then add regularization loss with a big factor
  – And check the gradient in each case
What if things are not working?

• Have a feeling of the initial loss value
  – For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is \(-\ln(0.1)=2.302\)
  – For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).

• Try to overfit on a tiny subset of the dataset
  – The cost should reach to zero if things are working properly
What if things are not working?

Learning rate might be too low; Batch size might be too small
What if things are not working?
What if things are not working?

- Plot the histogram of activations per layer
  - E.g., for tanh functions, we expect to see a diverse distribution of values between [-1,1]
What if things are not working?

- Visualize your layers (the weights)

Examples of visualized weights for the first layer of a neural network. **Left:** Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.
Andrew Ng’s suggestions

• “In DL, the coupling between bias & variance is weaker compared to other ML methods:
  – We can train a network to have high bias and variance.”

• “Dev (validation) and test sets should come from the same distribution. Dev&test sets are like problem specifications.
  – This requires especially attention if you have a lot of data from simulated environments etc. but little data from the real test environment.”

https://www.youtube.com/watch?v=F1ka6a13S9I
Andrew Ng’s suggestions

• “Knowing the human performance level gives information about the problem of your network:
  – If training error is far from human performance, then there is a bias error.
  – If they are close but validation has more error (compared to the diff between human and training error), then there is variance problem.”

• “After surpassing human level, performance increases only very slowly/difficult.
  – One reason: There is not much space for improvement (only tiny little details). Problem gets much harder.
  – Another reason: We get labels from humans.”

https://www.youtube.com/watch?v=F1ka6a13S9I
Also read the following

• 37 reasons why your neural network is not working:
  – https://medium.com/@slavivanov/4020854bd607
What is best then?

• Which algorithm to choose?
  – No answer yet
  – See Tom Schaul (2014)
  – RMSprop and AdaDelta seems to be slightly favorable; however, no best algorithm

• SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam are the most widely used ones
Luckily, deep networks are very powerful.

Regularization is turned off in the experiments. When you turn on regularization, the networks perform worse.
Concluding remarks for the first part

• Loss functions
• Gradients of loss functions for minimizing them
  – All operations in the network should be differentiable
• Gradient descent and its variants
• Initialization, normalization, adaptive learning rate, ...
• Overall, you have learned most of the tools you will use in the rest of the course.