ECE C147 Cheatsheet: Neural Networks and Deep Learning
Willson Luo

1 Basics

1.1 Linear Regression

Model the relationship as linear: $y = \theta^T x$ (assuming x includes bias term 1).

- **Loss Function (MSE):** We minimize the Mean Squared Error:

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{i=1}^{N}(y^{(i)} - \theta^T x^{(i)})^2 = \frac{1}{2N}||Y - X\theta||^2$$

- **Closed Form Solution:** Setting $\nabla_\theta \mathcal{L} = 0$ yields the Normal Equation (Least Squares):

$$\theta = (X^T X)^{-1} X^T Y$$

1.2 Generalization and model selection

Underfitting (High Bias)

- Model is too simple.
- High Training Error.
- High Testing Error.

Overfitting (High Variance)

- Model is too complex.
- Low Training Error.
- High Testing Error.

k-Fold Cross Validation: Split data into k folds. Train on $k-1$, validate on 1. Repeat k times and average error. Used when data is limited.

1.3 k-Nearest Neighbors (k-NN)

A non-parametric, "lazy" learning algorithm.

1. **Train:** Memorize all training data (no optimization).
2. **Predict:** Find the k training samples closest to new input x_{new} (usually L2 distance) and vote.

$$d(x^{(i)}, x^{(j)}) = ||x^{(i)} - x^{(j)}||_2 = \sqrt{\sum (x_k^{(i)} - x_k^{(j)})^2}$$

Hyperparameters: k (number of neighbors) and distance metric.

Pros/Cons:

- Fast training ($O(1)$), but slow testing ($O(N)$).
- **Curse of Dimensionality:** In high dimensions (e.g., images), distance becomes less meaningful, and "nearest" neighbors may not be close.

2 Softmax

We want to turn our output scores into probabilities.

$$\text{softmax}(z_i) = \frac{e^{a_i(x)}}{\sum_{j=1}^{c} e^{a_j(x)}}$$

for $a_i(x) = w_i^T x + b_i$ and c classes.

If we let $\theta = w_1, b_1, w_2, b_2, \ldots, w_c, b_c$ be the parameters of our model, then softmax can be interpreted as the probability x belongs to the class i.

$$\Pr(y^i = j | x^i, \theta) = \text{softmax}_j(x^i)$$

Intuitively, we want to optimize the parameters of our model so that our probabilities match up with measured probabilities. This is done by minimizing the cross-entropy loss function

$$\mathcal{L} = -\sum_{i=1}^{n}\sum_{j=1}^{c} z_j^i \log(\text{softmax}_j(x^i))$$

where z_j^i is the one-hot encoding of the class y^i.

3 Gradient descent

Gradient descent iteratively updates the parameters to minimze the loss function. Naively, we update the parameters as follows:

$$\theta = \theta - \epsilon \nabla_\theta \mathcal{L}$$

where ϵ is the learning rate. However, this can be computationally expensive, so we use stochastic gradient descent (SGD) or mini-batch instead. Mini-batch/SGD calculates the gradient at each step using a subset of the entire dataset.

4 Neural networks

The output of each layer is defined as

$$\mathrm{h} = f(Wx + b)$$

where f is the activation function that introduces non-linearity.

4.1 Activation functions

There are 3 main activation functions that others are based on.

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Pros:

- Linear behavior around $x = 0$.
- Differentiable everywhere.

Cons:

- Saturates at extremes (vanishing gradients).
- Non-negative (zig-zagging gradients).

Tanh

$$\tanh(x) = 2\sigma(2x) - 1$$

Pros:

- Linear behavior around $x = 0$.
- Differentiable everywhere.
- Zero-centered (smoother gradients).

Cons:

- Saturates at extremes.

ReLU

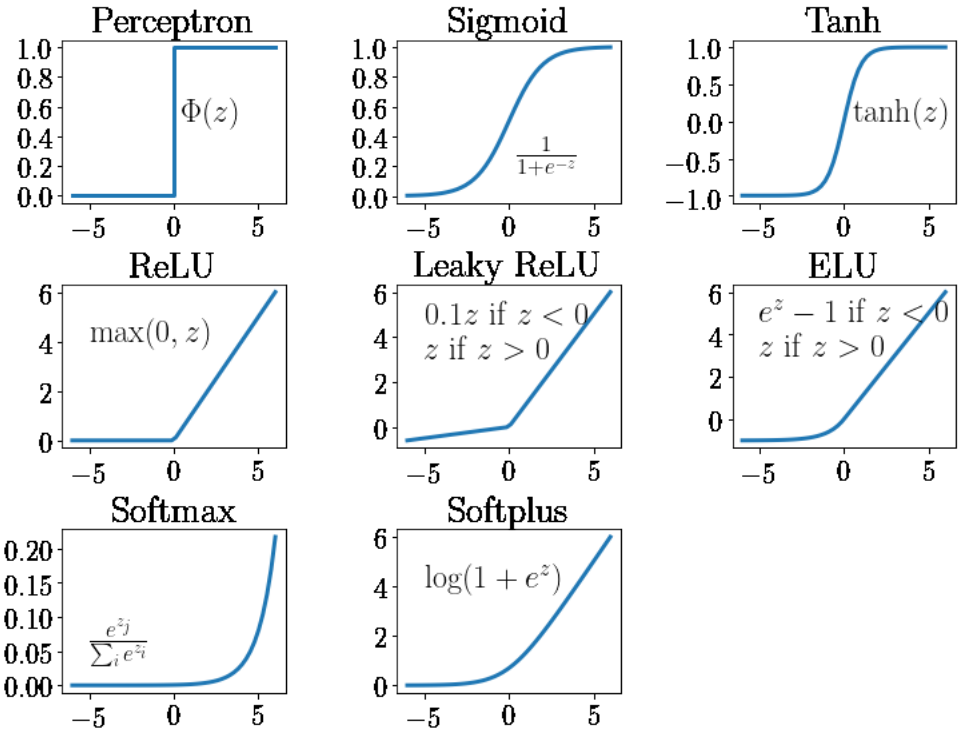
$$\mathrm{ReLU}(x) = \max(0, x)$$

Pros:

- Faster convergence.
- No saturation for $x > 0$.
- Active units behave linearly.

Cons:

- Non-negative.
- Dying ReLU problem (no learning for zero activation).



5 Backpropagation

Not all parameters are directly connected to the output, so we generalize the chain rule for derivatives to compute the gradients for all parameters.

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial z}{\partial \theta} \cdot \frac{\partial \mathcal{L}}{\partial z}$$

6 Initialization

A good initialization is essential so that your network can train effectively.

If we set our weights to be too small, then the outputs of each layer will be close to zero, which can lead to vanishing gradients. The gradient is multiplied by the weight at each layer, so if the weights are small, the gradient will become smaller and smaller as it propagates back through the layers.

If our weights are too large, we risk our gradients exploding, or in the case of sigmoid and tanh, saturating, which also leads to vanishing gradients.

6.1 Xavier and Kaiming initialization

Xavier initialization aims to keep the variance of the activations and gradients consistent across layers. Doing this ensures that the gradients don't vanish or explode as they propagate.

Concretely, the heuristics mean

$$\text{var}(h_i) = \text{var}(h_j) \quad \text{var}(\frac{\partial \mathcal{L}}{\partial h_i}) = \text{var}(\frac{\partial \mathcal{L}}{\partial h_j})$$

Solving for the variance of the forward and backward pass, we get

$$\text{var}(w) = \frac{1}{n_{in}} \quad \text{var}(w) = \frac{1}{n_{out}}$$

where n is the number of input or output units. We draw the weights from

$$\mathcal{N}(0, \frac{2}{n_{in} + n_{out}})$$

The Xavier initialization works well for sigmoid and tanh, but still leads to vanishing gradients for ReLU. This is because ReLU is zero for negative inputs, which effectively reduces the number of active neurons by half. To account for this we can use the He initialization, which draws weights from

$$\mathcal{N}(0, \frac{2}{n_{in}})$$

7 Batch normalization

An obstacle to standard training is that the distribution of the inputs to each layer changes as learning occurs in previous layers. This is called internal covariate shift, leading to variable activations. This greatly impacts the training speed and makes it harder to train deep networks.

Another consideration is that when we do gradient descent, we're calculating how to update parameters assuming the other layers are fixed. But these layers may change drastically.

Batch normalization makes the output of each layer have unit statistics (zero mean and unit variance). Learning then becomes simpler because the parameters in lower layers do not change the distribution of the inputs to higher layers.

Affine →BatchNorm →ReLu →Affine →BatchNorm →ReLu →Affine →Softmax

We first normalize the activations

$$\hat{x_i} = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

with

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^j \quad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_i^j - \mu_i)^2$$

It then passes through a learnable affine transformation allowing the model decide if it is better to not have unit statistics.

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

Empirically, batch normalization allows us to use higher learning rates, which can lead to faster convergence. It also acts as a regularizer, introducing some noise into the training process, which can help prevent overfitting.

8 Optimization

8.1 Regularization

Regularizations are used to improve model generalization by preventing overfitting. It can also be seen as increasing the estimator bias without increasing the variance.

8.1.1 L2 regularization

L2 regularization penalizes the size of the weights, promoting models with smaller weights.

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2}||w_i||^2$$

$$\nabla\mathcal{L} = \nabla\mathcal{L}_0 + \lambda w_i$$

8.1.2 L1 regularization

L1 regularization penalizes the absolute values of the weights, promoting sparsity (driving some weights to exactly zero).

$$\mathcal{L} = \mathcal{L}_0 + \lambda||w||_1$$

$$\nabla\mathcal{L} = \nabla\mathcal{L}_0 + \lambda\text{sign}(w)$$

8.2 Dataset augmentation

Dataset augmentation is a technique used to artificially increase the size of a training dataset by applying various transformations to the existing data. This can help improve the generalization of a model by exposing it to a wider variety of examples.

8.3 Multitask learning

Train a model to be able to perform multiple tasks. This represents the belief that multiple tasks share common factors to explain variations in data.

8.4 Transfer learning

Transfer learning is a technique where a model trained on one task is repurposed on a second related task. This is particularly useful when the second task has limited data, as the model can leverage the knowledge it has gained from the first task to improve performance on the second task. It is common to add a new linear layer at the end of the model and only train that layer, keeping the rest of the model fixed.

8.5 Ensemble methods

Train multiple different models and average their results at test time. The intuition is that different models will make different errors, so averaging their predictions can lead to better performance. We expect k independent models to have an error of $\frac{1}{k}\mathbb{E}\epsilon_i^2$

8.5.1 Bagging

Bagging (Bootstrap Aggregating) is a technique where multiple models are trained on different subsets of the training data. Each model is trained on a random sample of the data with replacement, and their predictions are averaged to produce the final output. In practice, neural networks already reach a wide variety of solutions given different random initializations.

8.6 Dropout

1. On a given training iteration, sample a binary mask for all input and hidden units, where each unit is kept with probability p and dropped with probability $1 - p$.
2. Apply the mask to all units, then perform a forward pass and backpropagation as usual.
3. At test time, use the full network but scale the activations by p to account for the fact that more units are active than during training.

8.6.1 Inverted dropout

Scale the activations by $\frac{1}{p}$ during training, so that at test time, we can use the full network without scaling. This is more efficient because we don't have to do any scaling at test time.

8.7 Momentum

We maintain the running mean of gradients and use that to update the parameters. This leads to finding more flat local minima, which is better for generalization.

$$v \leftarrow \alpha v - \epsilon g$$

$$\theta \leftarrow \theta + v$$

8.7.1 Nesterov momentum

The gradient is calculated at the parameter after taking a step

$$v \leftarrow \alpha v - \epsilon g(\theta + \alpha v)$$

Which is also equivalent to

$$\tilde{\theta}_{old} = \theta_{old} + \alpha v_{old}$$

$$v_{new} = \alpha v_{old} - \epsilon g(\tilde{\theta}_{old})$$

$$\tilde{\theta}_{new} = \tilde{\theta}_{old} + v_{new} + \alpha(v_{new} - v_{old})$$

8.7.2 Adagrad

Adagrad adapts the learning rate for each parameter based on the historical gradients.

$$a \leftarrow a + g \odot g$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{a} + \nu} \odot g$$

8.7.3 RMSProp

RMSProp is a modification of Adagrad that uses an exponentially weighted moving average. Fixes the problem of Adagrad strictly increasing the denominator.

$$a \leftarrow \beta a + (1 - \beta) g \odot g$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{a} + \nu} \odot g$$

8.7.4 Adam

Adam combines momentum and RMSProp. It maintains both a running average of the gradients and a running average of the squared gradients.

$$t \leftarrow t + 1$$

$$v \leftarrow \beta_1 v + (1 - \beta_1) g$$

$$a \leftarrow \beta_2 a + (1 - \beta_2) g \odot g$$

$$\hat{v} = \frac{v}{1 - \beta_1^t}$$

$$\hat{a} = \frac{a}{1 - \beta_2^t}$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\hat{a}} + \nu} \odot \hat{v}$$

9 Convolutional neural networks

Convolution essentially drags a filter or kernel across the input and computes the dot product at each location.

$$f \star g = \sum_{m=-\infty}^{\infty} f(m)g(n+m)$$

Convolution has sparse connections and reuses parameters. It prioritizes local patterns rather than looking at every single data point in isolation. The dimensions after applying a convolutional layer can be calculated as follows:

$$\left\lfloor \frac{w - w_f + 2p}{s} + 1 \right\rfloor \times \left\lfloor \frac{h - h_f + 2p}{s} + 1 \right\rfloor$$

More layers is generally required to capture global patterns as the farther down the network you go the more inputs each neuron is affected by. This is also known as the receptive field.x

9.1 Pooling layer

Filter which effectively downsamples the input. It is used to reduce the spatial dimensions of the input, which can help reduce the number of parameters and computational cost of the model. It also helps to make the model more robust to small translations in the input. Common pooling operations include max pooling and average pooling.

10 Appendix

10.1 Matrix operations

Matrix w.r.t. Vector	Matrix w.r.t. Matrix
$y = \mathbf{W}x$	$\mathbf{Y} = \mathbf{W}\mathbf{X}$
$\frac{\partial \mathcal{L}}{\partial x} = \mathbf{W}^T \frac{\partial \mathcal{L}}{\partial y}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \mathbf{W}^T \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial y} x^T$	$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \mathbf{X}^T$

$$\frac{\partial \ln \det(\mathbf{X})}{\partial \mathbf{X}} = (\mathbf{X}^{-1})^T = (\mathbf{X}^T)^{-1}$$

$$\frac{\partial \mathrm{tr}(\mathbf{X})}{\partial \mathbf{X}} = \mathbf{I}$$

10.2 Jacobian and Hessian

For $f : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian $J \in \mathbb{R}^{m \times n}$ is:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

For $f : \mathbb{R}^n \to \mathbb{R}$ (scalar), the Hessian $H \in \mathbb{R}^{n \times n}$ is:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

10.3 Local Gradient Patterns

Add Gate	**Multiply Gate**	**Max Gate**
$z = x + y$	$z = x \cdot y$	$z = \max(x, y)$
$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z} \cdot 1$	$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z} \cdot y$	$\frac{\partial \mathcal{L}}{\partial x} = \mathbb{1}(x > y) \frac{\partial \mathcal{L}}{\partial z}$
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z} \cdot 1$	$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z} \cdot x$	$\frac{\partial \mathcal{L}}{\partial y} = \mathbb{1}(y > x) \frac{\partial \mathcal{L}}{\partial z}$